About This Document

Purpose vii
Audience vii
Organization viii
Related Documentation ix
Conventions xi
  Syntax Notation xi
  Syntax Diagrams xii
  Keywords and Punctuation xiv
  Identifiers and Names xiv
Customer Support xv
  Support Solutions Warehouse xv
  General and Technical Questions xv
  Troubleshooting Tips xvii
  Documentation Questions and Comments xvii

1 Aroma—A Database for Decision Support
  Basic Aroma Schema 1-3
  Store, Market, and Promotion Dimensions 1-5
  About the Sales Facts 1-7
  Typical Data Warehousing Queries 1-9
  Summary 1-10

2 Basic Queries
  Retrieving Data: SELECT Statement 2-3
  Retrieving Specific Columns 2-5
  Retrieving Specific Rows: WHERE Clause 2-7
  Specifying Compound Conditions: OR, NOT, AND 2-9
  Specifying Complex Conditions 2-11
  Comparison Operators 2-13
  Comparison Predicates 2-15
  Using Wildcard Characters 2-17
  Joining Dimensions and Facts 2-19
3 Data Analysis

Using RISQL Display Functions 3-3
Cumulative Totals: CUME 3-5
Resetting Cumulative Totals: RESET BY Clause 3-7
Calculating Moving Averages: MOVINGAVG 3-9
Calculating Moving Sums: MOVINGSUM 3-11
Ranking Data: RANK 3-13
Ranking the Top Ten: RANK, WHEN 3-15
Ranking Values in Groups: NTILE 3-17
Ranking Values in Unequal Groups: CASE and NTILE 3-19
Ranking Values as High, Middle, or Low: TERTILE 3-21
Calculating Ratios as Percentages: RATIOTOREPORT*100 3-23
Incrementing or Decrementing Dates: DATEADD 3-25
Calculating Elapsed Days: DATEDIFF 3-27
Displaying Dateparts as Integers: EXTRACT 3-29
Summary 3-31

4 Comparison Queries

A Simple Comparison Query 4-3
A Solution for Comparing Data: CASE Expressions 4-5
A More Flexible Solution: Subqueries in the FROM Clause 4-7
Calculations with FROM Clause Subqueries 4-9
Comparisons with Select-List Subqueries 4-11
Correlated Subqueries in the Select List 4-13
Cross-References with Expressions 4-15
Calculations with Select-List Subqueries 4-17
Comparisons with WHERE Clause Subqueries 4-19
Comparison Predicates in Subqueries 4-21
EXISTS Predicate 4-23
SOME or ANY Predicate 4-25
Summary 4-26
5 Joins and Unions
   Inner Joins 5-3
   Joins in the FROM Clause 5-5
   Joining System Tables 5-7
   Joining a Table to Itself 5-9
   Outer Joins 5-11
   Left Outer Join 5-13
   Full Outer Join with ORDER BY, BREAK BY 5-15
   Combining Result Sets: UNION 5-17
   INTERSECT—Finding Common Rows 5-19
   INTERSECT of Fact Table Data 5-21
   EXCEPT—Finding the Exceptions in Two Result Sets 5-23
   Summary 5-24

6 Macros, Views, and Temporary Tables
   Basic Macros 6-3
   Embedded Macros 6-5
   Macros with Parameters 6-7
   Macros with Multiple Parameters 6-9
   Comparison Macros 6-11
   Share Comparison Macros 6-13
   Macros That Calculate Change in Share 6-15
   Selecting from Views 6-17
   Creating a Temporary Table 6-19
   Summary 6-20

A The Complete Aroma Database
   Basic Aroma Schema A-3
   The Class and Product Tables A-5
   The Market and Store Tables A-7
   The Period Table A-9
   The Promotion Table A-11
   The Sales Table A-13
   Purchasing Tables A-15
   The Supplier and Deal Tables A-17
   The Orders and Line_Items Tables A-19

Index
About This Document

Purpose

The goal of this SQL Self-Study Guide is to help query writers and application developers write commonly asked business questions by using standard SQL, RISQL® extensions, and RISQL macros. All of the queries used in examples are posed against the sample Aroma database, which is supplied with the Red Brick® Warehouse software.

Audience

The intended users of this guide are application developers, warehouse database administrators, and data analyst “power users” who write SQL statements. Knowledge of the RISQL Entry Tool, the RISQL Reporter, or some other interactive SQL tool is assumed. Familiarity with the structure of the SQL SELECT statement is also assumed.
About This Document

Organization

Most of this guide is formatted in a two-page style. The left page contains a business question, a SELECT statement that asks this question in a language your warehouse database understands, and a result. The facing right page gives a high-level description of the warehouse feature used in the example, describes some specific attributes of the example, and offers usage guidelines for writing similar queries.

This guide is divided into six chapters and an appendix:

• Chapter 1, "Aroma—A Database for Decision Support," presents the basic Aroma database, its tables, and its schema design. This chapter also discusses the kinds of questions that can be asked of such a database, given the RISQL functions and query expressibility of Red Brick Warehouse.

• Chapter 2, “Basic Queries,” reviews the SELECT statement and provides several standard SQL queries posed against the Aroma database. This chapter also shows how to use set functions (or aggregate functions) and introduces simple joins.

• Chapter 3, “Data Analysis,” shows how questions that require sequential processing can be easily expressed as queries with RISQL display functions. This chapter also presents some examples of datetime scalar functions.

• Chapter 4, “Comparison Queries,” illustrates the use of CASE expressions and subqueries as methods of comparing data and displaying the results in a spreadsheet format. The more complex examples show how to include calculations in comparison queries.

• Chapter 5, “Joins and Unions,” illustrates different kinds of table joins and shows how to use the UNION, INTERSECT, and EXCEPT operators.

• Chapter 6, “Macros, Views, and Temporary Tables,” shows how simple queries can be generalized and complex queries simplified with RISQL macros. The last two examples show how to create views and temporary tables to further simplify user interaction with warehouse data.

• Appendix A, “The Complete Aroma Database,” presents a more detailed picture of the Aroma database, including an additional purchasing schema that is used in a few examples in this guide.

Note: The examples in this guide build on each other and become more complex as the chapters progress; therefore, it is recommended that each chapter be read in sequence.
The standard documentation set for Red Brick Warehouse includes the following documents:

- **Installation and Configuration Guide**: Installation and configuration information, as well as platform-specific material, about Red Brick Warehouse and related products. Customized for either UNIX-based or Windows NT systems.

- **Warehouse Administrator’s Guide**: Description of warehouse architecture, supported schemas, and other concepts relevant to warehouse databases. Procedural information for designing and implementing a warehouse database, maintaining a database, and tuning a database for performance. Includes a description of the system tables and the configuration file (`rbw.config`). Customized for either UNIX-based or Windows NT systems.

- **Table Management Utility Reference Guide**: Description of the Table Management Utility, including all activities related to loading and maintaining data. Also includes information about data replication and the `rb_cm` copy management utility.


- **SQL Self-Study Guide**: Example-based review of SQL and introduction to the RISQL extensions, the macro facility, and Aroma, the sample database.

- **RISQL Entry Tool and RISQL Reporter User’s Guide**: Complete guide to the RISQL Entry Tool, a command-line tool used to enter SQL statements, and the RISQL Reporter, an enhanced version of the RISQL Entry Tool with report-formatting capabilities.

- **Messages and Codes Reference Guide**: Complete listing of all informational, warning, and error messages generated by warehouse products, including probable causes and recommended responses. Also includes event log messages that are written to the log files.

- **Release Notes**: Information pertinent to the current release that was unavailable when the documents were printed.
In addition to the standard documentation set, the following documents are included for specific sites:

**Red Brick Vista User’s Guide**
Description of the Red Brick Vista™ aggregate navigation and advice system, including procedures for rewriting queries and getting advice on the best set of aggregate tables and views to create. Includes detailed examples of queries whose performance can be dramatically increased by using aggregate navigation.

**SQL-BackTrack for Red Brick Warehouse User’s Guide**
The complete guide to SQL-BackTrack™ for Red Brick Warehouse, a command-line interface for backing up and recovering warehouse databases. Includes procedures for defining backup configuration files, performing online and checkpoint backups, and recovering the database to a consistent state.

**Client Connector Pack Installation Guide**
Procedures for installing and configuring the Red Brick ODBC Driver, the RISQL Entry Tool, and the RISQL Reporter on client systems. Included for those sites that purchase the Client Connector Pack.

**ODBC Connectivity Guide**
Information about ODBC conformance levels as well as instructions for compiling and linking an ODBC application using the Red Brick ODBClib SDK.

**Red Brick Data Mine User’s Guide**
Description of the data mining process, and procedural information for using the Red Brick Data Mine™ SQL-based interface to find hidden or unpredictable relationships among the data in a data set. Included for those sites that purchase the Red Brick Data Mine option.

**Red Brick Data Mine Builder™ User’s Guide**
Description of the data mining process, and procedural information for performing data mining using Red Brick’s GUI-based product in a Microsoft Windows environment.

Additional references you might find helpful include:
- An introductory-level book on SQL
- An introductory-level book on relational databases
- Documentation for your hardware platform and operating system

**Online Documentation**
The English version of the Red Brick Warehouse documentation is also available in Adobe Acrobat format (PDF) on a separate CD-ROM.
Conventions

Throughout Red Brick Systems technical publications, the following notation and syntax conventions are used:

- Computer input and output, including commands, code, and examples, appear in **Courier**.
- Information that you enter or that is being emphasized in an example appears in **Courier bold** to help you distinguish it from other text.
- Filenames, system-level commands, and variables appear in *Palatino italic* or **Courier italic**, depending on the context.
- Document titles always appear in *Palatino italic*.
- Names of database tables and columns are capitalized (Sales table, Dollars column). Names of system tables and columns are in all uppercase (RBW_INDEXES table, TNAME column).

Syntax Notation

This guide uses the following conventions to describe the syntax of operating-system commands:

<table>
<thead>
<tr>
<th>Command Element</th>
<th>Example</th>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values and parameters</td>
<td><code>table_name</code></td>
<td>Items that you replace with an appropriate name, value, or expression are in <em>italic</em> type style.</td>
</tr>
<tr>
<td>Optional items</td>
<td>[   ]</td>
<td>Optional items are enclosed by square brackets. Do not type the brackets.</td>
</tr>
<tr>
<td>Choices</td>
<td>ONE</td>
<td>TWO</td>
</tr>
<tr>
<td>Required choices</td>
<td>[ONE</td>
<td>TWO]</td>
</tr>
<tr>
<td>Default values</td>
<td>ONE</td>
<td>TWO</td>
</tr>
<tr>
<td>Repeating items</td>
<td>name, ...</td>
<td>Items that can be repeated are followed by a comma and an ellipsis. Separate the items with commas.</td>
</tr>
<tr>
<td>Language elements</td>
<td>( ) , ; .</td>
<td>Parentheses, commas, semicolons, and periods are language elements. Use them exactly as shown.</td>
</tr>
</tbody>
</table>
Syntax Diagrams

This guide uses diagrams built with the following components to describe the syntax for statements and all commands other than system-level commands:

<table>
<thead>
<tr>
<th>Component</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶️</td>
<td>Statement begins.</td>
</tr>
<tr>
<td>▶️</td>
<td>Statement syntax continues on next line. Syntax elements other than complete statements end with this symbol.</td>
</tr>
<tr>
<td>▶️</td>
<td>Statement continues from previous line. Syntax elements other than complete statements begin with this symbol.</td>
</tr>
<tr>
<td>▶️</td>
<td>Statement ends.</td>
</tr>
<tr>
<td>SELECT</td>
<td>Required item in statement.</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Optional item.</td>
</tr>
<tr>
<td>DBA TO</td>
<td>Required item with choice. One and only one item must be present.</td>
</tr>
<tr>
<td>CONNECT TO</td>
<td></td>
</tr>
<tr>
<td>SELECT ON</td>
<td></td>
</tr>
<tr>
<td>ASC</td>
<td>Optional item with choice. If a default value exists, it is printed in <strong>bold</strong>.</td>
</tr>
<tr>
<td>DESC</td>
<td></td>
</tr>
<tr>
<td>ASC</td>
<td>Optional items. Several items are allowed; a comma must precede each repetition.</td>
</tr>
<tr>
<td>DESC</td>
<td></td>
</tr>
</tbody>
</table>
The syntax elements shown above are combined to form a diagram as follows:

```
GRANT  ALL, PRIVILEGES  ON  table_name  
<table>
<thead>
<tr>
<th></th>
<th>DELETE</th>
<th>INSERT</th>
<th>SELECT</th>
<th>UPDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO</td>
<td>db_user_name, role_name, PUBLIC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Complex syntax diagrams such as the one for the following statement are repeated as point-of-reference aids for the detailed diagrams of their components. Point-of-reference diagrams are indicated by their shadowed corners, gray lines, and reduced size:

```
SELECT  select_list  from_clause  where_clause
|       | group_by_clause  | having_clause  | when_clause  |
```

The point-of-reference diagram is then followed by an expanded diagram of the shaded portion—in this case, the `select_list`:

```
ALL, DISTINCT  expression  *  
|       | c_alias  | AS  |
|       | table_name.* |
```
Keywords and Punctuation

Keywords are words reserved for statements and all commands except
system-level commands. When a keyword appears in a syntax diagram, it is
shown in uppercase. You can write a keyword in upper- or lowercase, but you
must spell the keyword exactly as it appears in the syntax diagram.

Any punctuation that occurs in a syntax diagram must also be included in
your statements and commands exactly as shown in the diagram.

Identifiers and Names

Metavariables serve as placeholders for identifiers and names in the syntax
diagrams and examples. A metavariable can be replaced by an arbitrary name,
identifier, or literal, depending on the context. Metavariables are also used to
represent complex syntax elements that are expanded in additional syntax
diagrams. When a metavariable appears in a syntax diagram, an example, or
text, it is shown in lowercase italic.

The following syntax diagram uses metavariables to illustrate the general form
of a simple SELECT statement:

```sql
SELECT <column_name> FROM <table_name>
```

When you write a SELECT statement of this form, you replace the
metavariables `column_name` and `table_name` with the name of a specific column
and table.
Customer Support

Please review the following information before contacting the Customer Support Center at Red Brick Systems.

Support Solutions Warehouse

The Support Solutions Warehouse is the Customer Support Center’s external web site, an online resource that registered Red Brick customers can use to:

- Submit new cases.
- Read release notes.
- Find answers to frequently asked questions (FAQs).
- Search the Problems and Solutions database.

To use the Support Solutions Warehouse, point your web browser to the following URL and enter your registered username and password:

http://www.redbrick.com/RBCustomer/index.htm

If you do not have a registered username and password, contact the Customer Support Center by telephone, fax, or e-mail.

General and Technical Questions

If you have general sales-related questions or technical questions about Red Brick products or services, contact Red Brick Systems as follows:

Telephone
General Questions (408) 399-3200 or 1 (800) 777-2585
Technical Questions (408) 399-7100 or 1 (800) 727-1866

FAX
General Questions (408) 399-3277
Technical Questions (408) 399-3297

Internet e-mail
General Questions info@redbrick.com
Technical Questions support@redbrick.com

World Wide Web www.redbrick.com
Existing Cases

If you want to inquire about the status of an existing case, please have the case number ready. The case number will always be given to you by the support engineer who logs the case or first contacts you. This number is used to keep track of all the activities performed during the resolution of each problem.

New Cases

If you want to log a new case, please have the following information ready:

- Red Brick Warehouse version
- Platform and operating-system version
- Error messages returned by Red Brick Warehouse or the operating system
- Concise description of the problem, including any commands or operations performed prior to the occurrence of the error message
- List of Red Brick Warehouse and/or operating-system configuration changes made prior to the occurrence of the error message

If you think the problem concerns client-server connectivity, please have the following additional information ready:

- Name and version of the client tool in use
- Version of Red Brick ODBC Driver in use (if applicable)
- Name and version of client network and/or TCP/IP stack in use
- Error messages returned by the client application
- Warehouse and client locale specifications
Troubleshooting Tips

You can often reduce the time it takes to close your case by providing the smallest possible reproducible example of your problem. The more you can isolate the cause of the problem, the more quickly the support engineer can help you resolve it.

- For SQL query problems, try removing columns or functions, or restating WHERE, ORDER BY, or GROUP BY clauses until you can isolate the part of the statement causing the problem.
- For TMU load problems, verify the datatype mapping between the source file and the target table to ensure compatibility. Try loading a small test set of data to determine whether the problem concerns volume or data format.
- For connectivity problems, verify that the network is up and running by issuing the `rbping` command from the client to the host. If possible, try another client tool to see if the same problem arises.

Documentation Questions and Comments

If you have questions or comments about the Red Brick Warehouse documentation, please contact the Technical Publications Department at Red Brick Systems as follows:

**Telephone**
+1 408 399 3200
+1 800 727 1866 (USA only)

**Internet e-mail**
docs@redbrick.com
This guide shows how to express commonly asked business questions as database queries by using the Structured Query Language (SQL) and the RISQL® extensions to SQL developed by Red Brick® Systems. This guide also illustrates how query writing can be simplified with RISQL macros when queries or parts of queries are issued repetitively.

All the examples in this document—and in most of the Red Brick Systems documentation—are based on Aroma, a sample database that contains sales data for coffee and tea products sold in stores across the United States. Each example consists of three parts:

- A business question, expressed in everyday language.
- One or more corresponding SELECT statements, expressed in SQL.
- A table of results returned from the database.

Aroma is typically installed when the Red Brick Warehouse software is installed. If you want to run the sample queries yourself, ask your warehouse administrator how to access the Aroma database at your site.

This chapter presents the tables of the basic Aroma database and briefly describes the primary key–foreign key relationships that link the data in these tables.

The final section of this chapter presents a few of the questions that the Aroma database—or any Red Brick Warehouse database—can answer quickly and efficiently.
Aroma—A Database for Decision Support

Aroma Database—Retail Schema

Most of the examples in this guide are based on data from the basic Aroma database, which tracks daily retail sales in stores owned by the Aroma Coffee and Tea Company. The retail schema consists of four main dimension tables—Period, Product, Store, and Promotion—and a Sales fact table, as well as two outboard tables, Class and Market.

The following figure illustrates this basic schema.

The crow’s-feet in this diagram indicate a one-to-many relationship between the two tables. For example, each distinct value in the Perkey column of the Period table can occur only once in that table but many times in the Sales table. Column names in bold are primary key columns. Column names in italic are foreign key columns. Column names in bold italic are primary and foreign key columns.

The remainder of this chapter presents sample data from each table so you can see how these primary key–foreign key relationships work.
Basic Aroma Schema

In a decision-support database, the tables and columns are named with familiar business terms, making the schema easy to understand and use. A well-designed schema provides the following benefits to application developers and end users:

- Business questions are easy to express as SQL queries.
- Queries run fast and return consistent answers.

The retail Aroma schema meets both of these criteria. The Sales table contains the everyday measurements of the business—the facts—and the Store, Period, Product, and Promotion tables contain the dimensions, or characteristics, of the business. The Class and Market tables contain information that adds another level of detail to the product and store information.

Most of the examples in this guide use the simple star schema formed by these seven basic tables, but the Aroma database also contains a purchasing schema with a more complex design. For details, refer to Appendix A, “The Complete Aroma Database.”

Note: The Aroma database does not contain any predefined aggregate tables. For information about using the Red Brick Vista™ query rewrite system to accelerate the performance of aggregate queries, refer to the Red Brick Vista User’s Guide.
Period, Product, and Class Dimensions

Period Table

Here are the first few rows of the Period table. The primary key column is the Perkey column.

<table>
<thead>
<tr>
<th>PERKEY</th>
<th>DATE</th>
<th>DAY</th>
<th>WEEK</th>
<th>MONTH</th>
<th>QTR</th>
<th>YEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1994-01-01</td>
<td>SA</td>
<td>1</td>
<td>JAN</td>
<td>Q1_94</td>
<td>1994</td>
</tr>
<tr>
<td>2</td>
<td>1994-01-02</td>
<td>SU</td>
<td>2</td>
<td>JAN</td>
<td>Q1_94</td>
<td>1994</td>
</tr>
<tr>
<td>3</td>
<td>1994-01-03</td>
<td>MO</td>
<td>2</td>
<td>JAN</td>
<td>Q1_94</td>
<td>1994</td>
</tr>
<tr>
<td>4</td>
<td>1994-01-04</td>
<td>TU</td>
<td>2</td>
<td>JAN</td>
<td>Q1_94</td>
<td>1994</td>
</tr>
<tr>
<td>5</td>
<td>1994-01-05</td>
<td>WE</td>
<td>2</td>
<td>JAN</td>
<td>Q1_94</td>
<td>1994</td>
</tr>
<tr>
<td>6</td>
<td>1994-01-06</td>
<td>TH</td>
<td>2</td>
<td>JAN</td>
<td>Q1_94</td>
<td>1994</td>
</tr>
</tbody>
</table>

Product and Class Tables

Here are the first few rows of the Product table. The primary key is a combination of the Classkey and Prodkey values.

<table>
<thead>
<tr>
<th>CLASSKEY</th>
<th>PRODKEY</th>
<th>PROD_NAME</th>
<th>PKG_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Veracruzano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Xalapa Lapa</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Expresso XO</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>La Antigua</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>Lotta Latte</td>
<td>No pkg</td>
</tr>
</tbody>
</table>

If a dimension table contains foreign key columns that reference other dimension tables, the referenced tables are called “outboard” or “outrigger” tables. For example, the Product table’s Classkey column is a foreign key reference to the Class table.

Here are the first few rows of the Class table:

<table>
<thead>
<tr>
<th>CLASSKEY</th>
<th>CLASS_TYPE</th>
<th>CLASS_DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bulk_beans</td>
<td>Bulk coffee products</td>
</tr>
<tr>
<td>2</td>
<td>Bulk_tea</td>
<td>Bulk tea products</td>
</tr>
<tr>
<td>3</td>
<td>Bulk_spice</td>
<td>Bulk spices</td>
</tr>
<tr>
<td>4</td>
<td>Pkg_coffee</td>
<td>Individually packaged coffee products</td>
</tr>
<tr>
<td>5</td>
<td>Pkg_tea</td>
<td>Individually packaged tea products</td>
</tr>
<tr>
<td>6</td>
<td>Pkg_spice</td>
<td>Individually packaged spice products</td>
</tr>
</tbody>
</table>
Aroma— A Database for Decision Support

Store, Market, and Promotion Dimensions

Dimension tables contain descriptions that data analysts use as they query the database. For example, the Store table contains store names and addresses; the Product table contains product and packaging information; and the Period table contains month, quarter, and year values. Every table contains a primary key that consists of one or more columns; each row in a table is uniquely identified by its primary key value or values.

Store and Market Tables

Here are the first few rows of the Store table (some columns have been truncated to fit on the page). The primary key column is Storekey; Mktkey is a foreign-key reference to the Market table.

<table>
<thead>
<tr>
<th>STOREKEY</th>
<th>MKTKEY</th>
<th>STORE_TYPE</th>
<th>STORE_NAME</th>
<th>STREET</th>
<th>CITY</th>
<th>STATE</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>Small</td>
<td>Roasters, Los Gatos</td>
<td>1234 University A</td>
<td>Los Gatos</td>
<td>CA</td>
<td>95032</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>Large</td>
<td>San Jose Roasting Co</td>
<td>5678 Bascom Ave</td>
<td>San Jose</td>
<td>CA</td>
<td>95156</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>Medium</td>
<td>Cupertino Coffee Sup</td>
<td>987 DeAnza Blvd</td>
<td>Cupertino</td>
<td>CA</td>
<td>97865</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Medium</td>
<td>Moulin Rouge Roastin</td>
<td>898 Main Street</td>
<td>New Orleans</td>
<td>LA</td>
<td>70125</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>Small</td>
<td>Moon Pennies</td>
<td>98675 University</td>
<td>Detroit</td>
<td>MI</td>
<td>48209</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>Small</td>
<td>The Coffee Club</td>
<td>9865 Lakeshore Bl</td>
<td>Chicago</td>
<td>IL</td>
<td>06060</td>
</tr>
</tbody>
</table>

Here are the first few rows of the Market table:

<table>
<thead>
<tr>
<th>MKTKEY</th>
<th>HQ_CITY</th>
<th>HQ_STATE</th>
<th>DISTRICT</th>
<th>REGION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Atlanta</td>
<td>GA</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>2</td>
<td>Miami</td>
<td>FL</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>3</td>
<td>New Orleans</td>
<td>LA</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>4</td>
<td>Houston</td>
<td>TX</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>5</td>
<td>New York</td>
<td>NY</td>
<td>New York</td>
<td>North</td>
</tr>
</tbody>
</table>

Promotion Table

Here are the first few rows of the Promotion table. The primary key column is Promokey.

<table>
<thead>
<tr>
<th>PROMOKEY</th>
<th>PROMO_TYPE</th>
<th>PROMO_DESC</th>
<th>VALUE</th>
<th>START_DATE</th>
<th>END_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>No promotion</td>
<td>0.00</td>
<td>9999-01-01</td>
<td>9999-01-01</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-01-01</td>
<td>1994-01-31</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-02-01</td>
<td>1994-02-28</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-03-01</td>
<td>1994-03-31</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-04-01</td>
<td>1994-04-30</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-05-01</td>
<td>1994-05-31</td>
</tr>
</tbody>
</table>
### Sales Table

Here are the first 20 rows of the Sales table.

<table>
<thead>
<tr>
<th>PERKEY</th>
<th>CLASSKEY</th>
<th>PRODKEY</th>
<th>STOREKEY</th>
<th>PROMOKEY</th>
<th>QUANTITY</th>
<th>DOLLARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>116</td>
<td>8</td>
<td>34.00</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>116</td>
<td>9</td>
<td>60.75</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>116</td>
<td>40</td>
<td>270.00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>30</td>
<td>1</td>
<td>116</td>
<td>16</td>
<td>36.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>22</td>
<td>1</td>
<td>116</td>
<td>11</td>
<td>30.25</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>30</td>
<td>1</td>
<td>116</td>
<td>30</td>
<td>187.50</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>116</td>
<td>25</td>
<td>143.75</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>12</td>
<td>87.00</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>115.50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>22</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>58.50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>101.50</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>22</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>99.00</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>46</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>36.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>10</td>
<td>40.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>36</td>
<td>279.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>132.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td>48.00</td>
</tr>
</tbody>
</table>

The primary key column is a combination of values from five columns:

- perkey
- classkey
- prodkey
- storekey
- promokey
About the Sales Facts

The Sales table is a fact table; the data it contains is easily accessible through the business attributes defined in the tables it references, and it stores large amounts of statistical information about those attributes. The Sales table is by far the largest table in the Aroma database and its data is split into two database storage areas (known as segments). For information about segments, refer to the Warehouse Administrator’s Guide.

Access to business facts must be easy and quick. Red Brick Warehouse provides such access by addressing fact table rows through business dimensions familiar to the query writer. For example, to retrieve sales of La Antigua coffee at the San Jose Roasting Company on January 31, 1996, you simply specify those three dimensions (1/31/96, product name, store name), and the warehouse server quickly retrieves your request.

Multi-Part Primary Key

The Sales table contains a multi-part primary key: Each of its five columns is a foreign key reference to another table’s primary key:

perkey, classkey, prodkey, storekey, promokey

This primary key links the Sales data to the Period, Product, Store, and Promotion dimensions. Through such links, figures regarding the sale of a specific product on a particular day in a given city—expressed in terms of dollars and quantities—can be quickly and easily retrieved from the database.
Commonly Asked Questions

Easy

• What were the weekly sales of Lotta Latte brand coffee in San Jose during last year?
• What were the average monthly sales of all coffee products in the West during each month of last year?

Moderately Difficult

• How do the sales of Lotta Latte in San Jose compare with its sales in Los Angeles and New York?
• How has the monthly market share of Lotta Latte changed during the last two years in all markets?
• Which suppliers charge the most for bulk tea products?
• What was the most successful promotion last December in California?

Very Difficult Without RISQL Extensions

• What were the running totals for Lotta Latte sales during each month of last year?
• What were the ratios of monthly to total sales (expressed as percentages) for Lotta Latte during the same period?
• Which ten cities had the worst coffee sales in 1995 with regard to dollar sales and quantities sold?
• Which Aroma stores fall into the top 25% in terms of sales revenue for the first quarter of 1996? Which stores fall into the middle 50%, and the bottom 25%?
**Typical Data Warehousing Queries**

Many kinds of commonly asked business questions can be readily expressed as SQL queries. For example, anyone familiar with SQL can write a query that returns the quarterly sales of a given product in a given year.

However, many other commonly asked questions cannot be expressed so easily. Questions that require comparisons often challenge both the query writers and SQL itself. For example, a question requesting a comparison of weekly, monthly, quarterly, and yearly values is one of the simplest questions posed during a sales analysis, but expressing this question as a query represents a formidable challenge to the query writer, the query language, and the database server.

Business questions that request sequential processing are very difficult to express as SQL queries. To derive a simple running total, for example, data analysts typically run several queries with a client tool, then paste the results together using another tool. This approach is awkward, because it requires a sophisticated user, floods the network with data, and takes place on a client that is typically much slower than a database server.

The RISQL extensions to SQL provide a better solution, because they are easy to use, reduce network traffic, and perform sequential calculations that execute quickly on the server.
Summary

This chapter briefly described the retail schema of the Aroma database and suggested some typical business questions that a Red Brick Warehouse database can answer.

A decision-support database is designed to be queried: It has a few easy-to-understand tables, provides exceptional query performance, and guarantees data integrity. To this end, the primary tables in a warehouse database are:

- Few in number
- Designed using the analyst’s vocabulary
- Reflective of the natural dimensions of the business

The remainder of this self-study guide consists of detailed examples that show how to write commonly asked business questions. Most of these examples are based on the Aroma retail schema. Some additional tables are used sporadically in the more advanced examples; these tables are described in Appendix A.
Basic Queries

Through a series of simple examples, this chapter illustrates how to retrieve data from a warehouse database with standard SQL SELECT statements.

This chapter describes how to:

- Retrieve specific columns and rows from a relational database table.
- Perform logical operations on retrieved data.
- Use wildcard characters in search conditions.
- Retrieve data from more than one table.
- Order data and calculate subtotals on numeric columns.
- Perform aggregate calculations with set functions.
- Group data.
- Perform arithmetic operations on retrieved data.
- Remove rows from the result set if specified columns contain NULLs, zeroes, or spaces.
Basic Queries

SELECT Statement

Question
What regions, districts, and markets are defined in the Aroma database?

Query

```sql
select * from market;
```

Result

<table>
<thead>
<tr>
<th>MKTKEY</th>
<th>HQ_CITY</th>
<th>HQ_STATE</th>
<th>DISTRICT</th>
<th>REGION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Atlanta</td>
<td>GA</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>2</td>
<td>Miami</td>
<td>FL</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>3</td>
<td>New Orleans</td>
<td>LA</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>4</td>
<td>Houston</td>
<td>TX</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>5</td>
<td>New York</td>
<td>NY</td>
<td>New York</td>
<td>North</td>
</tr>
<tr>
<td>6</td>
<td>Philadelphia</td>
<td>PA</td>
<td>New York</td>
<td>North</td>
</tr>
<tr>
<td>7</td>
<td>Boston</td>
<td>MA</td>
<td>Boston</td>
<td>North</td>
</tr>
<tr>
<td>8</td>
<td>Hartford</td>
<td>CT</td>
<td>Boston</td>
<td>North</td>
</tr>
<tr>
<td>9</td>
<td>Chicago</td>
<td>IL</td>
<td>Chicago</td>
<td>Central</td>
</tr>
<tr>
<td>10</td>
<td>Detroit</td>
<td>MI</td>
<td>Chicago</td>
<td>Central</td>
</tr>
<tr>
<td>11</td>
<td>Minneapolis</td>
<td>MN</td>
<td>Minneapolis</td>
<td>Central</td>
</tr>
<tr>
<td>12</td>
<td>Milwaukee</td>
<td>WI</td>
<td>Minneapolis</td>
<td>Central</td>
</tr>
<tr>
<td>14</td>
<td>San Jose</td>
<td>CA</td>
<td>San Francisco</td>
<td>West</td>
</tr>
<tr>
<td>15</td>
<td>San Francisco</td>
<td>CA</td>
<td>San Francisco</td>
<td>West</td>
</tr>
<tr>
<td>16</td>
<td>Oakland</td>
<td>CA</td>
<td>San Francisco</td>
<td>West</td>
</tr>
<tr>
<td>17</td>
<td>Los Angeles</td>
<td>CA</td>
<td>Los Angeles</td>
<td>West</td>
</tr>
<tr>
<td>19</td>
<td>Phoenix</td>
<td>AZ</td>
<td>Los Angeles</td>
<td>West</td>
</tr>
</tbody>
</table>
Retrieving Data: SELECT Statement

You use SELECT statements to retrieve columns and rows of data from database tables, to perform arithmetic operations on the data, and to group and/or order the data. In most cases, a SELECT statement contains a simple query expression that begins with the SELECT keyword and is followed by one or more clauses and subclauses. (For detailed information about more complex query expressions, refer to Chapter 7 of the SQL Reference Guide.)

The most basic SELECT statement contains two keywords—SELECT and FROM:

```
SELECT select_list
FROM table_list;
```

where:

- `select_list`  Column names or SQL expressions separated by commas. An asterisk (*) can also be used.
- `table_list`  Table names separated by commas. Referenced tables must contain the column names in `select_list`.

SELECT and FROM (and all other words shown in uppercase in subsequent references to syntax in this guide) are reserved SQL keywords. These words must be used exactly as defined by the SQL standard. SQL is not case-sensitive, so keywords can be written in uppercase or lowercase.

About the Example

This example query retrieves the entire contents of the Market table. The asterisk symbol (*) is the SQL abbreviation for “all column names that occur in `table_list`.” All column names in the Market table could be listed instead.

Red Brick Warehouse also supports explicit tables, whereby this query could be stated simply as:

```
table market;
```

Usage Notes

Names in a select list must be defined in tables listed in the FROM clause; exceptions to this rule are discussed later in this chapter. Columns are returned from the database in the order listed. If you use an asterisk, columns are returned as stored in the database table.

The semicolon (;) at the end of each example in this guide is not part of SQL syntax; it is an end-of-statement marker required by the RISQL Entry Tool and the RISQL Reporter. Depending on the interactive SQL tool you use to enter queries, you might not need to specify such a marker.
Basic Queries

SELECT List

Question
Which districts and regions are defined in the Aroma database?

Query
select district, region
from market;

Result
DISTRICT     REGION
Atlanta       South
Atlanta       South
New Orleans   South
New Orleans   South
New York      North
New York      North
Boston        North
Boston        North
Chicago       Central
Chicago       Central
Minneapolis  Central
Minneapolis  Central
San Francisco West
San Francisco West
San Francisco West
Los Angeles   West
Los Angeles   West
**Basic Queries**

**Retrieving Specific Columns**

By naming the columns in the select list of a SELECT statement, you can retrieve a specific set of columns from any table. Columns are returned in the order specified in the select list.

**About the Example**

This example requests a list of districts and their corresponding regions from the Market table.

**Usage Notes**

Although column names in the select list must be defined in the tables referenced in the FROM clause, other expressions can also occur in the select list. Several examples of such expressions are discussed later in this guide.

When the select list does not include all the columns in a table, a query might return duplicate rows, as in the example on the facing page. You can eliminate the duplicates by using the DISTINCT keyword. For example, the following query returns only the names of distinct districts and regions in the Market table.

```sql
select distinct district, region
from market;
```

<table>
<thead>
<tr>
<th>DISTRICT</th>
<th>REGION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>Boston</td>
<td>North</td>
</tr>
<tr>
<td>Chicago</td>
<td>Central</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>West</td>
</tr>
<tr>
<td>Minneapolis</td>
<td>Central</td>
</tr>
<tr>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>New York</td>
<td>North</td>
</tr>
<tr>
<td>San Francisco</td>
<td>West</td>
</tr>
</tbody>
</table>
WHERE Clause

Question

What products are sold without packaging?

Query

```sql
select prod_name, pkg_type
from product
where pkg_type = 'No pkg';
```

Result

<table>
<thead>
<tr>
<th>prod_name</th>
<th>pkg_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Veracruzano</td>
<td>No pkg</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>No pkg</td>
</tr>
<tr>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>No pkg</td>
</tr>
<tr>
<td>La Antigua</td>
<td>No pkg</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>No pkg</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>No pkg</td>
</tr>
<tr>
<td>NA Lite</td>
<td>No pkg</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>No pkg</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>No pkg</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>No pkg</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>No pkg</td>
</tr>
<tr>
<td>Special Tips</td>
<td>No pkg</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>No pkg</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Ruby’s Allspice</td>
<td>No pkg</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>No pkg</td>
</tr>
<tr>
<td>Travel Mug</td>
<td>No pkg</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>No pkg</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>No pkg</td>
</tr>
</tbody>
</table>
Retrieving Specific Rows: WHERE Clause

By including a set of logical conditions in a query, you can retrieve a specific set of rows from a table. Logical conditions are declared in the WHERE clause. If a row satisfies the conditions, the query returns the row; if not, the row is discarded. Logical conditions are also called search conditions, predicates, constraints, or qualifications.

The WHERE Clause

\[
\text{SELECT select_list} \\
\text{FROM table_list} \\
\quad \text{[WHERE search_condition]};
\]

where search_condition is a condition that evaluates to true or false.

The square brackets ([ ]) indicate that the WHERE clause is optional.

About the Example

This example query retrieves and displays the names of products that are not pre-packed or packaged. The warehouse server evaluates the condition

\[ \text{pkg_type} = 'No pkg' \]

for each row of the Product table and returns only those rows that satisfy the condition.

Usage Notes

A character literal is a character string enclosed within single quotes. To represent a single quote in a character literal, use two single quotes (" "). For example:

'Scarlet O'Hara'

Character literals must be expressed as stored in the database—in either uppercase or lowercase. For example, the server evaluates the condition

\[ \text{class_type} = 'Bulk_beans' \]

as false when the referenced column contains the string

'BULK_beans'

Set functions are not allowed in the WHERE clause. For more information about set functions, refer to page 2-25.
OR Connective

**Question**

What cities and districts are located in the southern or western regions?

**Query**

```sql
select hq_city, district, region
from market
where region = 'South' or region = 'West';
```

**Result**

<table>
<thead>
<tr>
<th>HQ_CITY</th>
<th>DISTRICT</th>
<th>REGION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>New Orleans</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>Houston</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>San Jose</td>
<td>San Francisco</td>
<td>West</td>
</tr>
<tr>
<td>San Francisco</td>
<td>San Francisco</td>
<td>West</td>
</tr>
<tr>
<td>Oakland</td>
<td>San Francisco</td>
<td>West</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>Los Angeles</td>
<td>West</td>
</tr>
<tr>
<td>Phoenix</td>
<td>Los Angeles</td>
<td>West</td>
</tr>
</tbody>
</table>
**Specifying Compound Conditions: OR, NOT, AND**

Most queries written for decision-support analysis contain compound conditions. Compound conditions are simple conditions joined by logical connectives. SQL contains the following logical connectives:

<table>
<thead>
<tr>
<th>Connective</th>
<th>Name</th>
<th>Order of Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Parentheses (force order of evaluation)</td>
<td>1</td>
</tr>
<tr>
<td>NOT</td>
<td>Negation</td>
<td>2</td>
</tr>
<tr>
<td>AND</td>
<td>And</td>
<td>3</td>
</tr>
<tr>
<td>OR</td>
<td>Or</td>
<td>4</td>
</tr>
</tbody>
</table>

The server evaluates compound conditions as follows: all the NOT operators first, all the AND connectives second, and all the OR connectives last. This evaluation order is commonly known as the *order of precedence*.

You can control the order of evaluation by grouping compound conditions with parentheses. In a nest of parentheses, the warehouse server evaluates the innermost set of parentheses first, the next innermost set next, and so on. Whenever the logic of a compound condition is not obvious, make it obvious with parentheses.

**About the Example**

This query retrieves all cities and districts in the southern or western regions. All rows that have *South* or *West* in their Region column satisfy the compound condition and are returned in the result table.

**Usage Notes**

When in doubt about the order of evaluation, force the order by grouping conditions with parentheses.
**Basic Queries**

**AND Connective**

**Question**

Which large or small Aroma stores are located in Los Angeles or San Jose?

**Query**

```sql
select store_type, store_name, city
from store
where (store_type = 'Large' or store_type = 'Small')
and (city = 'Los Angeles' or city = 'San Jose');
```

**The Result**

<table>
<thead>
<tr>
<th>STORE_TYPE</th>
<th>STORE_NAME</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large</td>
<td>San Jose Roasting Company</td>
<td>San Jose</td>
</tr>
<tr>
<td>Large</td>
<td>Beaches Brew</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>Small</td>
<td>Instant Coffee</td>
<td>San Jose</td>
</tr>
</tbody>
</table>
Specifying Complex Conditions

Search conditions—especially those written for decision-support analysis—can become quite complex; though constructed from simple conditions that use the logical connectives AND, OR, and NOT, complex conditions might be difficult to understand. Fortunately, SQL is free-form, so the logical structure of these conditions can be shown by using tab characters, blanks, and newline characters to define white space and logical relationships.

About the Example

This query retrieves and displays the names of Aroma stores that are both large or small and located in Los Angeles or San Jose.

The parentheses in this query are essential, because the AND connective has a higher precedence than the OR connective. If you remove the parentheses, the query returns a different result table:

<table>
<thead>
<tr>
<th>STORE_TYPE</th>
<th>STORE_NAME</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large</td>
<td>San Jose Roasting Company</td>
<td>San Jose</td>
</tr>
<tr>
<td>Large</td>
<td>Beaches Brew</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>Small</td>
<td>Instant Coffee</td>
<td>San Jose</td>
</tr>
<tr>
<td>Large</td>
<td>Miami Espresso</td>
<td>Miami</td>
</tr>
<tr>
<td>Large</td>
<td>Olympic Coffee Company</td>
<td>Atlanta</td>
</tr>
</tbody>
</table>

Usage Notes

A query retrieves and displays any data that is not explicitly excluded by its search condition, and a query with only a few general conditions can return an enormous number of rows.

Whenever you doubt how the server might evaluate a compound condition, explicitly group the conditions with parentheses to force the order of evaluation.
Basic Queries

>, <= Operators

Question
Which cities and districts are identified by Mktkey values that are greater than 4 and less than or equal to 12?

Query
```
select mktkey, hq_city, hq_state, district
from market
where mktkey > 4
  and mktkey <= 12;
```

Result
```
<table>
<thead>
<tr>
<th>MKTKEY</th>
<th>HQ_CITY</th>
<th>HQ_STATE</th>
<th>DISTRICT</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>New York</td>
<td>NY</td>
<td>New York</td>
</tr>
<tr>
<td>6</td>
<td>Philadelphia</td>
<td>PA</td>
<td>New York</td>
</tr>
<tr>
<td>7</td>
<td>Boston</td>
<td>MA</td>
<td>Boston</td>
</tr>
<tr>
<td>8</td>
<td>Hartford</td>
<td>CT</td>
<td>Boston</td>
</tr>
<tr>
<td>9</td>
<td>Chicago</td>
<td>IL</td>
<td>Chicago</td>
</tr>
<tr>
<td>10</td>
<td>Detroit</td>
<td>MI</td>
<td>Chicago</td>
</tr>
<tr>
<td>11</td>
<td>Minneapolis</td>
<td>MN</td>
<td>Minneapolis</td>
</tr>
<tr>
<td>12</td>
<td>Milwaukee</td>
<td>WI</td>
<td>Minneapolis</td>
</tr>
</tbody>
</table>
```
Comparison Operators

Conditions evaluate to true or false, and can be expressed with comparison operators or comparison predicates. Comparison predicates are described on the next two pages.

SQL contains the following comparison operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
</tbody>
</table>

About the Example

This example query retrieves and displays all cities and districts whose Mktkey is greater than 4 but less than or equal to 12.

The Mktkey column contains integer values, which are comparable to other numeric values. If you compare an integer to a character, however, the server returns an error message:

```sql
select mktkey, hq_city, hq_state, district
from market
where mktkey > '4';
```

** ERROR ** (19) Operands of comparison must have comparable datatypes.

Usage Notes

Conditions must compare values of comparable datatypes. If you attempt to compare unlike datatypes, the server returns either an error message or an incorrect result. Comparison operators can be used to compare one character string with another, as the following legal condition illustrates:

```sql
(city > 'L')
```

For more information about comparable datatypes, refer to Chapter 2 of the SQL Reference Guide.
Basic Queries

IN Predicate

Question

What cities are in the Chicago, New York, and New Orleans districts?

Query

```sql
select hq_city, hq_state, district
from market
where district in
    ('Chicago', 'New York', 'New Orleans');
```

Result

<table>
<thead>
<tr>
<th>HQ_CITY</th>
<th>HQ_STATE</th>
<th>DISTRICT</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Orleans</td>
<td>LA</td>
<td>New Orleans</td>
</tr>
<tr>
<td>Houston</td>
<td>TX</td>
<td>New Orleans</td>
</tr>
<tr>
<td>New York</td>
<td>NY</td>
<td>New York</td>
</tr>
<tr>
<td>Philadelphia</td>
<td>PA</td>
<td>New York</td>
</tr>
<tr>
<td>Chicago</td>
<td>IL</td>
<td>Chicago</td>
</tr>
<tr>
<td>Detroit</td>
<td>MI</td>
<td>Chicago</td>
</tr>
</tbody>
</table>
Comparison Predicates

A simple condition can be expressed with the following SQL comparison predicates:

<table>
<thead>
<tr>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN expression1 AND expression2</td>
</tr>
<tr>
<td>LIKE pattern</td>
</tr>
<tr>
<td>IN (list)</td>
</tr>
<tr>
<td>IS NULL</td>
</tr>
<tr>
<td>IS NOT NULL</td>
</tr>
<tr>
<td>ALL</td>
</tr>
<tr>
<td>SOME or ANY</td>
</tr>
<tr>
<td>EXISTS</td>
</tr>
</tbody>
</table>

Examples of the ALL, SOME/ANY, and EXISTS predicates are presented in Chapter 4, "Comparison Queries."

For syntax descriptions and examples of all these predicates, as well as detailed definitions of simple and complex expressions, refer to Chapter 3 of the SQL Reference Guide.

About the Example

This query lists all the cities in the Chicago, New York, and New Orleans districts. It could also be written with the equals comparison operator (=) and a set of OR conditions:

```sql
where district = 'Chicago'
  or district = 'New York'
  or district = 'New Orleans'
```

Usage Notes

Strive to write logical sets of conditions that are simple, easy to understand, and easy to maintain. Always clarify the logical structure of your compound conditions with ample white space, define logical blocks by indentation, and force evaluation precedence with parentheses.
Basic Queries

% Wildcard

Question

Which cities are in districts that begin with the letters Min?

Query

```sql
select district, hq_city
from market
where district like 'Min%';
```

Result

<table>
<thead>
<tr>
<th>DISTRICT</th>
<th>HQ_CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minneapolis</td>
<td>Minneapolis</td>
</tr>
<tr>
<td>Minneapolis</td>
<td>Milwaukee</td>
</tr>
</tbody>
</table>
Using Wildcard Characters

Previous queries have expressed conditions that match complete character strings. With the LIKE predicate and the two wildcard characters (\%, \_), you can also express conditions that match a portion of a character string (a substring).

The percent wildcard matches any character string. For example:
- like 'TOT%' is true for any string that begins with 'TOT'.
- like '%FRESH' is true for any string that ends with 'FRESH'.
- like '%ZERO%' is true for any string that contains 'ZERO'.

The percent sign can also be used to search for a null character string—zero characters.

The underscore wildcard matches any one character in a fixed position. For example:
- like '_EE_' is true for any four-letter string whose two middle characters are 'EE'.
- like '%LE_N%' is true for any string that contains the pattern 'LE_N'. The strings 'CLEAN', 'KLEEN', and 'VERY KLEEN' all match this pattern.

About the Example

This query retrieves the names of all districts that begin with the characters Min and lists the cities in these districts. The wildcard percent sign (%) allows for any character combination (including blank spaces) after the n in Min, but what comes before the n must match the character pattern exactly as stored.

Usage Notes

A LIKE condition is TRUE when its pattern matches a substring in a column. If the pattern contains no wildcard characters, the pattern must match the column entry exactly. For example, the condition

    month like 'APRIL'

is true only when the column entry contains the character string APRIL and nothing else. In other words, this condition is equivalent to

    month = 'APRIL'

The LIKE predicate can be used only on columns that contain character strings.
**Basic Queries**

**Simple Joins**

**Question**

What were the daily sales totals for Easter products sold on weekends on a type 900 promotion in 1995 and which stores registered those sales?

**Query 1**

```sql
select prod_name, store_name, day, dollars
from promotion, product, period, store, sales
where promotion.promokey = sales.promokey
  and product.prodkey = sales.prodkey
  and product.classkey = sales.classkey
  and period.perkey = sales.perkey
  and store.storekey = sales.storekey
  and prod_name like 'Easter%'
  and day in ('SA', 'SU')
  and promo_type = 900
  and year = 1995;
```

**Query 2**

```sql
select prod_name, store_name, day, dollars
from promotion natural join sales
  natural join product
  natural join period
  natural join store
where prod_name like 'Easter%'
  and day in ('SA', 'SU')
  and promo_type = 900
  and year = 1995;
```

**Two Queries—Same Result**

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>STORE_NAME</th>
<th>DAY</th>
<th>DOLLARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easter Sampler Basket</td>
<td>Olympic Coffee Company</td>
<td>SU</td>
<td>30.00</td>
</tr>
<tr>
<td>Easter Sampler Basket</td>
<td>Miami Espresso</td>
<td>SA</td>
<td>270.00</td>
</tr>
</tbody>
</table>
Joining Dimensions and Facts

So far, the queries in this chapter have retrieved data from a single table; however, most queries join information from different tables. Typically, dimension tables are joined to fact tables to constrain the facts in interesting ways. For example, you can join the Sales fact table to its Store and Product dimensions to get sales figures per product per store, or to its Period and Product dimensions to get sales figures per product per week.

About the Example

This business question requires a join of five tables in the Aroma retail schema: the Sales fact table and its Product, Period, Store, and Promotion dimensions. To join tables in a query, you must give the database server explicit instructions on how to perform the join. In Query 1, the joins are specified in the WHERE clause with five simple conditions that join the Sales table to its dimensions over its five primary key columns. The Product table has a two-part primary key, so it is joined to the Sales table over two columns: Prodkey and Classkey. Because all of these conditions involve identically named joining columns, the question can alternatively be posed in the style of Query 2, using natural joins in the FROM clause. This approach to joining tables works well with the Aroma database because its main tables form a simple star schema and all the foreign key columns use the same names as the primary keys they reference.

Natural joins operate on all pairs of identically named columns shared by the tables; therefore, in Query 2 the Sales and Product tables are joined over both the Classkey column and the Prodkey column.

There are two other ways to join these tables in the FROM clause and get the same result; for details, refer to Chapter 5, “Joins and Unions,” which identifies the different types of join queries you can write and presents more examples.

Usage Notes

Any two tables can be joined over columns with comparable datatypes; joins are not dependent on the primary key–foreign key relationships used in this example.
**Basic Queries**

**ORDER BY Clause**

**Question**

What were the sales figures for Assam Gold Blend and Earl Grey at the Instant Coffee store during November 1995? Order the figures for each product from highest to lowest.

**Query**

```
select prod_name, store_name, dollars
from store natural join sales
    natural join product
    natural join period
where (prod_name like 'Assam Gold%' 
  or prod_name like 'Earl%')
  and store_name like 'Instant%'
  and month = 'NOV'
  and year = 1995
order by prod_name, dollars desc;
```

**Result**

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>STORE_NAME</th>
<th>DOLLARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>96.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>78.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>66.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>58.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>58.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>39.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>39.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>32.50</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>48.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>45.50</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>42.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>32.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>24.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Instant Coffee</td>
<td>20.00</td>
</tr>
</tbody>
</table>
**Ordering the Result Table: ORDER BY Clause**

You can use the ORDER BY clause to sort the result table of a query by the values in one or more specified columns. The default sort order is ascending (ASC); the DESC keyword changes the sort order to descending for the specified column, as shown in the example on the facing page.

To order results by an expression in the select list (for example, a set function), specify a column alias for the expression and then name the alias in the ORDER BY clause. For more information about column aliases, refer to page 2-27.

**ORDER BY clause**

```sql
SELECT select_list
FROM table_list
[WHERE search_condition]
[ORDER BY order_list];
```

where `order_list` is a list of columns by which data is ordered. Columns in the `order_list` need not occur in the `select_list`, but must exist in tables referenced in the FROM clause.

**About the Example**

This example query retrieves Assam Gold Blend and Earl Grey sales figures at the Instant Coffee store during November 1995. The query sorts the results by product and total daily sales.

**Usage Notes**

The ORDER BY clause must follow the other clauses in the SELECT statement (except the SUPPRESS BY clause) and include a list of columns to be ordered. A column can be referenced by its name, column alias, or position (ordinal number) in the select list. For example, the ORDER BY clause on the facing page could be written as follows:

```sql
order by prod_name, 3 desc
```

By specifying columns in `order_list` that are not in the `select_list`, you can order data by columns that are not displayed in the result table.
Calculating Subtotals

Question

What were the daily sales and monthly subtotals for Assam Gold Blend, Darjeeling Special, and Earl Grey teas at the Instant Coffee store during November 1995? What is the monthly subtotal for all three products?

Query

```
select prod_name, store_name, dollars
from store natural join sales
    natural join product
    natural join period
where prod_name in ('Assam Gold Blend', 'Earl Grey', 'Darjeeling Special')
    and store_name like 'Instant%'
    and month = 'NOV'
    and year = 1995
order by prod_name, dollars desc
break by prod_name summing 3;
```

Result

```
PROD_NAME                      STORE_NAME                     DOLLARS
Assam Gold Blend               Instant Coffee                   96.00
Assam Gold Blend               Instant Coffee                   78.00
Assam Gold Blend               Instant Coffee                   66.00
Assam Gold Blend               Instant Coffee                   58.50
Assam Gold Blend               Instant Coffee                   58.50
Assam Gold Blend               Instant Coffee                   39.00
Assam Gold Blend               Instant Coffee                   39.00
Assam Gold Blend               Instant Coffee                   32.50
Assam Gold Blend               NULL                            467.50
Darjeeling Special             Instant Coffee                  207.00
Darjeeling Special             Instant Coffee                  168.00
Darjeeling Special             Instant Coffee                  149.50
Darjeeling Special             Instant Coffee                  144.00
Darjeeling Special             Instant Coffee                  138.00
Darjeeling Special             Instant Coffee                  132.00
Darjeeling Special             Instant Coffee                   96.00
Darjeeling Special             Instant Coffee                   69.00
Darjeeling Special             Instant Coffee                   60.00
Darjeeling Special             Instant Coffee                   60.00
Darjeeling Special             Instant Coffee                   48.00
Darjeeling Special             NULL                            1271.50
Earl Grey                      Instant Coffee                   48.00
Earl Grey                      Instant Coffee                   45.50
Earl Grey                      Instant Coffee                   42.00
Earl Grey                      Instant Coffee                   32.00
Earl Grey                      Instant Coffee                   24.00
Earl Grey                      Instant Coffee                   20.00
Earl Grey                      NULL                            211.50
NULL                           NULL                            1950.50
```
Calculating Subtotals: BREAK BY Clause

When a query contains an ORDER BY clause, you can use a BREAK BY clause to add control breaks to the result set and calculate subtotals on numeric columns. The BREAK BY clause also computes a grand total of the subtotals and displays this value in the final row of the report. This clause is a RISQL extension to the ANSI SQL-92 standard.

**BREAK BY Clause**

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[ORDER BY order_list]
[BREAK BY order_reference SUMMING select_reference_list];
```

where `order_reference` is a column used in the `order_list` and `select_reference_list` is a numeric expression used in the `select_list`.

**About the Example**

The example query lists the daily totals for three tea products at the Instant Coffee store in November 1995. A subtotal of the sales figures is calculated for each product, and the grand total for all three products is displayed at the end of the report. The `order_reference` is Prod_Name and the `select_reference_list` consists of a single column reference (3, which refers to the Dollars column).

**Usage Notes**

As well as performing simple aggregate calculations on ordered sets of rows, the BREAK BY clause makes the contents of a long result set easier to read and absorb.

If the query contains a RISQL display function, the ORDER BY clause can contain another RISQL extension—the RESET BY subclause. For more details, refer to page 3-7.

A query that includes a BREAK BY clause cannot be used as a query expression in an INSERT INTO...SELECT statement.
Basic Queries

**SUM, AVG, MAX, MIN, COUNT**

*Question*

What were the total Lotta Latte sales figures in Los Angeles for 1995? What were the average, maximum, and minimum daily sales figures for that year, and how many daily totals were counted to produce these aggregate values?

*Query*

```sql
SELECT SUM(dollars), AVG(dollars), MAX(dollars), MIN(dollars),
       COUNT(*)
FROM store
     NATURAL JOIN sales
     NATURAL JOIN period
     NATURAL JOIN product
WHERE prod_name LIKE 'Lotta Latte%'
   AND year = 1995
   AND city LIKE 'Los Ang%';
```

*Result*

```
13706.50    171.33125000    376.00     39.00          80
```
Using Set Functions

Set functions operate on groups of values. For example, `SUM(dollars)` calculates the total dollars returned in a result table, and `AVG(dollars)` returns the average. The SQL set functions listed in the following table can occur one or more times in the select list.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM(expression)</td>
<td>Calculates the sum of all the values in expression.</td>
</tr>
<tr>
<td>SUM(DISTINCT expression)</td>
<td>Calculates the sum of distinct values in expression.</td>
</tr>
<tr>
<td>AVG(expression)</td>
<td>Calculates the average of all the values in expression.</td>
</tr>
<tr>
<td>AVG(DISTINCT expression)</td>
<td>Calculates the average of distinct values in expression.</td>
</tr>
<tr>
<td>MAX(expression)</td>
<td>Determines the maximum value in expression.</td>
</tr>
<tr>
<td>MIN(expression)</td>
<td>Determines the minimum value in expression.</td>
</tr>
<tr>
<td>COUNT(*)</td>
<td>Counts the number of rows returned.</td>
</tr>
<tr>
<td>COUNT(expression)</td>
<td>Counts the number of non-null values in expression.</td>
</tr>
<tr>
<td>COUNT(DISTINCT expression)</td>
<td>Counts the number of distinct non-null values in expression.</td>
</tr>
</tbody>
</table>

You can replace `expression` with any column name or numeric expression. Each function, except `COUNT(*)`, ignores NULL values when calculating the returned aggregate value. The `DISTINCT` keyword can occur only once in the select list.

About the Example

This query retrieves sales figures for Lotta Latte in Los Angeles during 1995. The result set also includes the average, maximum, and minimum sales during the year, and the number of daily totals on which those calculations are based.

Usage Notes

If the result set will contain individual values as well as aggregate values, the query must contain a GROUP BY clause. For more information about the GROUP BY clause, refer to page 2-29.

Set functions can be used as arguments to RISQL display functions; however, display functions cannot be used as arguments to set functions. (Display functions are discussed in Chapter 3, “Data Analysis.”)
Column Aliases

Question
What were the annual Lotta Latte sales figures in Los Angeles stores during 1995? What were the average, maximum, and minimum sales figures during this time period, and how many daily totals were counted in these aggregate values? Specify headings for the aggregate result columns.

Query
```
select sum(dollars) as dol_sales, avg(dollars) as avg_sales, 
    max(dollars) as max_dol, min(dollars) as min_dol, 
    count(*) as num_items 
from store natural join sales 
    natural join period 
    natural join product 
where prod_name like 'Lotta Latte%' 
    and year = 1995 
    and city like 'Los Ang%';
```

Result
```
DOL_SALES    AVG_SALES    MAX_DOL    MIN_DOL    NUM_ITEMS
13706.50     171.3312500 376.00    39.00      80
```
Using Column Aliases: AS

By default, the SELECT command returns values calculated by set functions but does not label the returned values with headings. You can specify a label, or column alias, for any column with the keyword AS followed by a character string. This alias can then be referenced in later clauses in the query.

For example, the following AS clause assigns the alias Dol_Sales to the Dollars column:

```
dollars as dol_sales
```

Column aliases are most useful when used to reference expressions from the select list in later clauses, as shown on page 2-29.

About the Example

This example query returns the same result set as the previous query in this chapter; however, in this case, column aliases are assigned to create headings for the aggregated results.

Usage Notes

Improve the readability of your result tables by assigning column aliases to all set functions that occur in your query select list.

An alias is a database identifier; it must begin with a letter and have a maximum length of 128 characters. Zero or more letters, digits, or underscores can follow the initial letter. A keyword cannot serve as a database identifier. For more details, refer to Chapter 2 of the SQL Reference Guide.

A column alias can occur anywhere in a SELECT statement to designate the column to which it refers (for example, in a WHERE, ORDER BY, GROUP BY, or HAVING clause).

Note: If the value contained in the column referenced by the column alias is the result of a set function, it cannot occur in the WHERE clause; however, it may occur in the HAVING clause.
Basic Queries

**GROUP BY**

Question

What were the annual totals for sales of coffee mugs in 1994 in each district? What were the average, maximum, and minimum sales during this time period? List the results by district.

Query

```sql
select district as district_city, sum(dollars) as dol_sales,
    avg(dollars) as avg_sales, max(dollars) as max_sales,
    min(dollars) as min_sales
from market natural join store
    natural join sales
    natural join period
    natural join product
where prod_name like '%Mug%'
    and year = 1994
  group by district_city
order by dol_sales desc;
```

Result

<table>
<thead>
<tr>
<th>DISTRICT_CITY</th>
<th>DOL_SALES</th>
<th>AVG_SALES</th>
<th>MAX_SALES</th>
<th>MIN_SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>1378.30</td>
<td>35.34102564</td>
<td>98.55</td>
<td>4.00</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>711.60</td>
<td>30.93913043</td>
<td>98.55</td>
<td>9.95</td>
</tr>
<tr>
<td>San Francisco</td>
<td>410.45</td>
<td>25.65312500</td>
<td>54.75</td>
<td>5.00</td>
</tr>
</tbody>
</table>
Grouping Rows: GROUP BY Clause

Set functions operate on all rows of a result table or on groups of rows defined by a GROUP BY clause. For example, you can group the sales for each market and calculate the respective sum, maximum, and minimum values.

GROUP BY Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[ORDER BY order_list]
[BREAK BY order_reference SUMMING select_reference_list];
```

where `group_list` is a list of column names (either in `select_list` or in tables listed in the FROM clause) or column aliases in `select_list`. All non-aggregated columns in `select_list` must appear in `group_list`.

About the Example

The example query retrieves annual sales totals for coffee mugs in 1994 (they are sold in three districts only), ordering the figures from highest to lowest. Conceptually speaking, the warehouse server processes this query as follows:

1. Retrieves all rows of data from tables specified in the FROM clause, joins the rows from separate tables, and generates an intermediate result table.
2. Retains all rows from the intermediate result table that satisfy the search condition specified in the WHERE clause.
3. Divides the result table into groups specified in the GROUP BY clause.
4. Processes all set functions on specified groups for the entire result table.
5. Orders results according to the ORDER BY clause.
6. Returns only those columns specified in the select list.

Usage Notes

You can accelerate the performance of aggregate queries—queries that contain set functions and/or GROUP BY clauses—by using the Red Brick Vista query rewrite system. For details, refer to the Red Brick Vista User’s Guide.

An ORDER BY clause references items in the select list by column name, column alias, or position. However, if the item in the order list is a set function, it must be referenced by its alias (Dol_Sales) or position number because it has no column name. For more information about column aliases, refer to page 2-27.
Basic Queries

Multiple Groups

Question

What were the total sales in each city during 1994 and 1995? List city names by year within their region and district.

Query

```
select year, region, district, city, sum(dollars) as sales
from market natural join store
natural join sales
natural join product
natural join period
where year in (1994, 1995)
group by year, region, district, city
order by year, region, district, city;
```

Result

<table>
<thead>
<tr>
<th>YEAR</th>
<th>REGION</th>
<th>DISTRICT</th>
<th>CITY</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1994</td>
<td>Central</td>
<td>Chicago</td>
<td>Chicago</td>
<td>133462.75</td>
</tr>
<tr>
<td>1994</td>
<td>Central</td>
<td>Chicago</td>
<td>Detroit</td>
<td>135023.50</td>
</tr>
<tr>
<td>1994</td>
<td>Central</td>
<td>Minneapolis</td>
<td>Milwaukee</td>
<td>172321.50</td>
</tr>
<tr>
<td>1994</td>
<td>North</td>
<td>Boston</td>
<td>Boston</td>
<td>184647.50</td>
</tr>
<tr>
<td>1994</td>
<td>North</td>
<td>Boston</td>
<td>Hartford</td>
<td>69196.25</td>
</tr>
<tr>
<td>1994</td>
<td>North</td>
<td>New York</td>
<td>New York</td>
<td>181735.00</td>
</tr>
<tr>
<td>1994</td>
<td>North</td>
<td>New York</td>
<td>Philadelphia</td>
<td>172395.75</td>
</tr>
<tr>
<td>1994</td>
<td>South</td>
<td>Atlanta</td>
<td>Atlanta</td>
<td>230346.45</td>
</tr>
<tr>
<td>1994</td>
<td>South</td>
<td>Atlanta</td>
<td>Miami</td>
<td>220519.75</td>
</tr>
<tr>
<td>1994</td>
<td>South</td>
<td>New Orleans</td>
<td>Houston</td>
<td>183853.75</td>
</tr>
<tr>
<td>1994</td>
<td>South</td>
<td>New Orleans</td>
<td>New Orleans</td>
<td>193052.25</td>
</tr>
<tr>
<td>1994</td>
<td>West</td>
<td>Los Angeles</td>
<td>Los Angeles</td>
<td>219397.20</td>
</tr>
<tr>
<td>1994</td>
<td>West</td>
<td>Los Angeles</td>
<td>Phoenix</td>
<td>192605.25</td>
</tr>
<tr>
<td>1994</td>
<td>West</td>
<td>San Francisco</td>
<td>Cupertino</td>
<td>180088.75</td>
</tr>
<tr>
<td>1994</td>
<td>West</td>
<td>San Francisco</td>
<td>Los Gatos</td>
<td>176992.75</td>
</tr>
<tr>
<td>1994</td>
<td>West</td>
<td>San Francisco</td>
<td>San Jose</td>
<td>398829.10</td>
</tr>
<tr>
<td>1994</td>
<td>Central</td>
<td>Chicago</td>
<td>Detroit</td>
<td>135023.50</td>
</tr>
<tr>
<td>1994</td>
<td>Central</td>
<td>Minneapolis</td>
<td>Milwaukee</td>
<td>172395.75</td>
</tr>
<tr>
<td>1994</td>
<td>Central</td>
<td>Minneapolis</td>
<td>Minneapolis</td>
<td>132125.75</td>
</tr>
<tr>
<td>1994</td>
<td>North</td>
<td>Boston</td>
<td>Boston</td>
<td>189761.00</td>
</tr>
<tr>
<td>1994</td>
<td>North</td>
<td>Boston</td>
<td>Hartford</td>
<td>135879.50</td>
</tr>
<tr>
<td>1994</td>
<td>North</td>
<td>New York</td>
<td>New York</td>
<td>171749.75</td>
</tr>
<tr>
<td>1994</td>
<td>North</td>
<td>New York</td>
<td>Philadelphia</td>
<td>171759.50</td>
</tr>
<tr>
<td>1994</td>
<td>South</td>
<td>Atlanta</td>
<td>Atlanta</td>
<td>229615.05</td>
</tr>
<tr>
<td>1994</td>
<td>South</td>
<td>Atlanta</td>
<td>Miami</td>
<td>234458.90</td>
</tr>
<tr>
<td>1994</td>
<td>South</td>
<td>New Orleans</td>
<td>Houston</td>
<td>186394.25</td>
</tr>
<tr>
<td>1994</td>
<td>South</td>
<td>New Orleans</td>
<td>New Orleans</td>
<td>190441.75</td>
</tr>
<tr>
<td>1995</td>
<td>West</td>
<td>Los Angeles</td>
<td>Los Angeles</td>
<td>228433.00</td>
</tr>
<tr>
<td>1995</td>
<td>West</td>
<td>Los Angeles</td>
<td>Phoenix</td>
<td>197044.50</td>
</tr>
<tr>
<td>1995</td>
<td>West</td>
<td>San Francisco</td>
<td>Cupertino</td>
<td>196439.75</td>
</tr>
<tr>
<td>1995</td>
<td>West</td>
<td>San Francisco</td>
<td>Los Gatos</td>
<td>175048.75</td>
</tr>
<tr>
<td>1995</td>
<td>West</td>
<td>San Francisco</td>
<td>San Jose</td>
<td>398829.10</td>
</tr>
</tbody>
</table>
Basic Queries

Nesting Grouped Results: GROUP BY Clause

When several column names occur in a GROUP BY clause, the result table is divided into groups within groups. For example, if you specify column names for year, region, and district in the GROUP BY clause, the returned figures are divided by year, each year is divided by region, and each region is divided by district.

GROUP BY Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[ORDER BY order_list]
[BREAK BY order_reference SUMMING select_reference_list];
```

where `group_list` is a list of column names (in either the `select_list` or the tables in the `table_list`) or column aliases in the `select_list`. Columns that do not participate in a set function (non-aggregate columns) in the `select_list` must appear in the `group_list`.

About the Example

The example query retrieves annual sales of all products for each city during 1994 and 1995. The sales figures are both grouped and ordered by year, region, district, and city.

Note: The cities referred to in this query are the city locations of each store, as defined in the Store table, not the cities defined as “hq_cities” in the Market table.

Usage Notes

If the select list includes an aggregation function but the query has no GROUP BY clause, all column references must be aggregation functions.
Basic Queries

Division Operator (/)

Question

What was the average price per sale of each product during 1994? Calculate the average as the total sales dollars divided by the total sales quantity.

Query

```
select prod_name, sum(dollars) as total_sales,
     sum(quantity) as total_qty,
     string(sum(dollars)/sum(quantity), 7, 2) as price
from product natural join sales
     natural join period
where year = 1994
group by prod_name
order by price;
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>TOTAL_SALES</th>
<th>TOTAL_QTY</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold Tips</td>
<td>38913.75</td>
<td>11563</td>
<td>3.36</td>
</tr>
<tr>
<td>Special Tips</td>
<td>38596.00</td>
<td>11390</td>
<td>3.38</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>41137.00</td>
<td>11364</td>
<td>3.61</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>39205.00</td>
<td>10767</td>
<td>3.64</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>42295.50</td>
<td>10880</td>
<td>3.88</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>44381.00</td>
<td>10737</td>
<td>4.13</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>48759.00</td>
<td>11094</td>
<td>4.39</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>1054.00</td>
<td>213</td>
<td>4.94</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>62283.25</td>
<td>11539</td>
<td>5.39</td>
</tr>
<tr>
<td>Ruby's Allspice</td>
<td>133188.50</td>
<td>23444</td>
<td>5.68</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>71419.00</td>
<td>11636</td>
<td>6.13</td>
</tr>
<tr>
<td>Colombiano</td>
<td>188474.50</td>
<td>27548</td>
<td>6.84</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>203544.00</td>
<td>28344</td>
<td>7.18</td>
</tr>
<tr>
<td>La Antigua</td>
<td>197069.50</td>
<td>26826</td>
<td>7.34</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>201230.00</td>
<td>26469</td>
<td>7.60</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>224020.00</td>
<td>28558</td>
<td>7.84</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>15395.35</td>
<td>1953</td>
<td>7.88</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>217994.50</td>
<td>25994</td>
<td>8.07</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>213510.00</td>
<td>26340</td>
<td>8.10</td>
</tr>
<tr>
<td>Aroma Sounds Cassette</td>
<td>5206.00</td>
<td>620</td>
<td>8.39</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>251590.00</td>
<td>29293</td>
<td>8.58</td>
</tr>
<tr>
<td>NA Lite</td>
<td>231845.00</td>
<td>25884</td>
<td>8.95</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>282385.25</td>
<td>28743</td>
<td>9.82</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>20278.50</td>
<td>1870</td>
<td>10.84</td>
</tr>
<tr>
<td>Travel Mug</td>
<td>1446.35</td>
<td>133</td>
<td>10.87</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>127207.00</td>
<td>10931</td>
<td>11.63</td>
</tr>
<tr>
<td>Spice Sampler</td>
<td>6060.00</td>
<td>505</td>
<td>12.00</td>
</tr>
<tr>
<td>Aroma Sounds CD</td>
<td>7125.00</td>
<td>550</td>
<td>12.95</td>
</tr>
<tr>
<td>French Press, 2-Cup</td>
<td>3329.80</td>
<td>224</td>
<td>14.86</td>
</tr>
<tr>
<td>Spice Jar</td>
<td>4229.00</td>
<td>235</td>
<td>17.99</td>
</tr>
<tr>
<td>French Press, 4-Cup</td>
<td>3323.65</td>
<td>167</td>
<td>19.90</td>
</tr>
<tr>
<td>Tea Sampler</td>
<td>13695.00</td>
<td>550</td>
<td>24.90</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Arithmetic Operators: ( ), +, -, *, /**

You can perform arithmetic operations within a select list or within a search condition. A complete set of arithmetic operators is listed in the following table. The order of evaluation precedence is from highest to lowest (top to bottom) and, within a given level, left to right, in the table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Forces order of evaluation</td>
</tr>
<tr>
<td>+, -</td>
<td>Positive and negative</td>
</tr>
<tr>
<td>*, /</td>
<td>Multiplication and division</td>
</tr>
<tr>
<td>+, -</td>
<td>Addition and subtraction</td>
</tr>
</tbody>
</table>

If you have any doubt about the order of evaluation for a given expression, group the expression with parentheses. For example, the server evaluates (4 + 3 * 2) as 10 but evaluates the grouped expression ((4 + 3) * 2) as 14.

**Usage Notes**

This query would normally return long-numeric values for the Price column. The STRING scalar function is used to remove all but two of the decimal places from each Price value:

```sql
string((sum(dollars)/sum(quantity)), 7, 2) as price
```

For more information about the STRING function and other scalar functions, refer to Chapter 5 of the *SQL Reference Guide*.
Basic Queries

HAVING Clause

Question
Which products had total sales of less than $25,000 during 1995? How many individual sales were made?

Query
```
select prod_name, sum(dollars) as total_sales, 
    sum(quantity) as total_units 
from product natural join sales 
    natural join period 
where year = 1995 
group by prod_name 
having total_sales < 25000 
order by total_sales desc;
```

Result
```
<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>TOTAL_SALES</th>
<th>TOTAL_UNITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma t-shirt</td>
<td>21397.65</td>
<td>1967</td>
</tr>
<tr>
<td>Espresso Machine Royale</td>
<td>18119.80</td>
<td>324</td>
</tr>
<tr>
<td>Espresso Machine Italiano</td>
<td>17679.15</td>
<td>177</td>
</tr>
<tr>
<td>Coffee Sampler</td>
<td>16634.00</td>
<td>557</td>
</tr>
<tr>
<td>Tea Sampler</td>
<td>14907.00</td>
<td>597</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>13437.20</td>
<td>1696</td>
</tr>
<tr>
<td>Aroma Sheffield Steel Teapot</td>
<td>8082.00</td>
<td>270</td>
</tr>
<tr>
<td>Spice Sampler</td>
<td>7788.00</td>
<td>649</td>
</tr>
<tr>
<td>Aroma Sounds CD</td>
<td>5937.00</td>
<td>459</td>
</tr>
<tr>
<td>Aroma Sounds Cassette</td>
<td>5323.00</td>
<td>630</td>
</tr>
<tr>
<td>French Press, 4-Cup</td>
<td>4570.50</td>
<td>230</td>
</tr>
<tr>
<td>Spice Jar</td>
<td>4073.00</td>
<td>227</td>
</tr>
<tr>
<td>French Press, 2-Cup</td>
<td>3042.75</td>
<td>205</td>
</tr>
<tr>
<td>Travel Mug</td>
<td>1581.75</td>
<td>145</td>
</tr>
<tr>
<td>Easter Sampler Basket</td>
<td>1500.00</td>
<td>50</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>1258.00</td>
<td>256</td>
</tr>
<tr>
<td>Christmas Sampler</td>
<td>1230.00</td>
<td>41</td>
</tr>
</tbody>
</table>
```
Conditions on Groups: HAVING Clause

Although dividing data into groups reduces the amount of information returned, queries often still return more information than you need. You can use a HAVING clause to exclude groups that fail to satisfy a specified condition, such as sums of dollars that are less than or higher than a given number.

This query calculates the total sales revenue for each product in 1995, then retains only those products whose totals fall below $25,000.

HAVING Clause

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[HAVING condition]
[ORDER BY order_list]
[BREAK BY order_reference SUMMING select_reference_list];
```

where `condition` is an SQL condition that can include set functions.

The HAVING clause differs from the WHERE clause in the following ways:

<table>
<thead>
<tr>
<th>WHERE Clause</th>
<th>HAVING Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Works on rows of data prior to grouping.</td>
<td>Works on the result set after grouping.</td>
</tr>
<tr>
<td>Conditions cannot be expressed with set functions (for example, SUM or AVG), but column aliases for non-aggregate expressions can be used.</td>
<td>Conditions can be expressed with any set function or column alias.</td>
</tr>
</tbody>
</table>

Usage Notes

Any set function can be referenced in a condition in the HAVING clause. A query with a HAVING clause must contain a GROUP BY clause unless the select list contains only set functions. For example:

```
select min(prodkey), max(classkey)
from product
having min(prodkey) = 0;
```
Basic Queries

Removing Rows That Contain NULLs, Zeroes, and Spaces

Question

What is the average discount applied to orders received from each Aroma supplier?

Query 1

```sql
select name as supplier,
       dec(sum(discount)/count(order_no),7,2) as avg_deal
from supplier natural join orders
     natural join deal
group by supplier
order by avg_deal desc;
```

Result

<table>
<thead>
<tr>
<th>SUPPLIER</th>
<th>AVG_DEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Espresso Express</td>
<td>500.00</td>
</tr>
<tr>
<td>Western Emporium</td>
<td>66.66</td>
</tr>
<tr>
<td>Aroma West Mfg.</td>
<td>50.00</td>
</tr>
<tr>
<td>CB Imports</td>
<td>47.50</td>
</tr>
<tr>
<td>Leaves of London</td>
<td>40.00</td>
</tr>
<tr>
<td>Tea Makers, Inc.</td>
<td>20.00</td>
</tr>
<tr>
<td>Colo Coffee</td>
<td>0.00</td>
</tr>
<tr>
<td>Aroma East Mfg.</td>
<td>0.00</td>
</tr>
<tr>
<td>Crashing By Design</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Query 2

```sql
select name as supplier,
       dec(sum(discount)/count(order_no),7,2) as avg_deal
from supplier natural join orders
     natural join deal
group by supplier
order by avg_deal desc
    suppress by 2;
```

Result

<table>
<thead>
<tr>
<th>SUPPLIER</th>
<th>AVG_DEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Espresso Express</td>
<td>500.00</td>
</tr>
<tr>
<td>Western Emporium</td>
<td>66.66</td>
</tr>
<tr>
<td>Aroma West Mfg.</td>
<td>50.00</td>
</tr>
<tr>
<td>CB Imports</td>
<td>47.50</td>
</tr>
<tr>
<td>Leaves of London</td>
<td>40.00</td>
</tr>
<tr>
<td>Tea Makers, Inc.</td>
<td>20.00</td>
</tr>
</tbody>
</table>

2-36   SQL Self-Study Guide
Removing Blank Rows: SUPPRESS BY Clause

If one or more columns in the data retrieved by a query contain NULLs, spaces, or zeroes, you can use a SUPPRESS BY clause to remove those rows from the final result set. This clause is a RISQL extension to the ANSI SQL-92 standard.

**SUPPRESS BY Clause**

```sql
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[HAVING condition]
[ORDER BY order_list]
[BREAK BY order_reference SUMMING select_reference_list]
[SUPPRESS BY column_list];
```

where `column_list` is a list of column names or aliases from the `select_list`, or a list of positional numbers that specify those columns.

**About the Example**

The first example query retrieves a complete list of Aroma suppliers, whether or not they have given discounts on orders; consequently, the result set lists three suppliers whose average deal amounts to zero dollars.

The second example removes those three suppliers from the result set by suppressing rows that contain 0.00 in column 2 (Avg_Deal).

**Usage Notes**

Note the use of the DEC scalar function to truncate the long-numeric values for the Avg_Deal column. Unlike the STRING function, which is described on page 2-33, this function converts the average values to more precise decimal values (not character strings).

The SUPPRESS BY clause is applied before any RISQL display functions in the query are computed; consequently, you cannot suppress rows by referencing a column that contains a display function. For examples of queries that include display functions, refer to Chapter 3, “Data Analysis.”
Basic Queries

Summary

The SELECT Statement

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[HAVING search_condition]
[ORDER BY order_list]
    [BREAK BY order_reference SUMMING select_reference_list]
    [SUPPRESS BY column_list];
```

Logical Connectives

( )  parentheses (force order of evaluation)

NOT  negation

AND  and

OR   or

Comparison Operators  Comparison Predicates

=  equal  BETWEEN  expression1 AND expression2

<  less than  LIKE  pattern

>  greater than  IN  (list)

<>  not equal  IS  NULL

>=  greater than or equal  IS  NOT NULL

<=  less than or equal
Basic Queries

Summary

This chapter discussed how to express many commonly asked business questions as SELECT statements and how to retrieve, group, and order data selected from one or more relational tables. This chapter also showed how to perform aggregate calculations such as sums, averages, minimums, and maximums, how to calculate subtotals with the BREAK BY clause, and how to use the SUPPRESS BY clause to remove rows that contain zeroes, NULLs, or space characters.

Most questions discussed in this chapter are easily expressed as standard SELECT statements and challenge neither the user nor SQL. The remaining chapters of this guide address more difficult questions, questions that require sequential processing, comparisons of aggregated values, more complex join specifications, or lengthy SELECT statements.

The next chapter shows how you can use RISQL extensions to answer business questions that require sequential processing.
Basic Queries
This chapter describes how to write queries that require some kind of data analysis. Many of the queries contain sequential calculations, or calculations that operate on an ordered set of rows—queries frequently encountered during business analysis. For example:

- What is the cumulative total (or running sum) by month?
- What is the moving average by week?
- How do monthly sales figures rank with one another?
- What is the ratio of this month’s sales to annual sales?

Standard SQL cannot perform these types of calculations, so analysts typically retrieve all the data required for the calculation and then perform sequential calculations with a client tool—a process that can be laborious and time consuming. Red Brick Systems provides RISQL display functions as a solution to this problem. With these functions, sequential calculations are performed quickly and easily on the server and only the results you want are returned to your data-analysis application.

This chapter also shows how to use scalar functions to calculate and extract date information from DATETIME columns.
### RISQL Display Functions

<table>
<thead>
<tr>
<th>Function Name and Syntax</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUME(expression)</td>
<td>Calculate a cumulative sum (running total).</td>
</tr>
<tr>
<td>MOVINGAVG(expression, n)</td>
<td>Calculate an average of the previous $n$-rows.</td>
</tr>
<tr>
<td>MOVINGSUM(expression, n)</td>
<td>Calculate a sum of the previous $n$ rows.</td>
</tr>
<tr>
<td>NTILE(expression, n)</td>
<td>Determine an $n$-level rank of values.</td>
</tr>
<tr>
<td>RANK(expression)</td>
<td>Determine a numeric rank of values.</td>
</tr>
<tr>
<td>RATIOTOREPORT(expression)</td>
<td>Calculate a ratio of portion to total.</td>
</tr>
<tr>
<td>TERTILE(expression)</td>
<td>Determine a three-level (high, middle, and low) rank of values.</td>
</tr>
</tbody>
</table>
RISQL display functions operate on sets of rows and perform sequential calculations. For example, the function CUME(dollars) returns a cumulative total of dollars for each row of a result table.

Within a SELECT statement, RISQL display functions can be used:

- In the select list
- In an expression
- As arguments of scalar functions
- As a condition in a WHEN clause
- In a subquery

RISQL display functions cannot be used:

- As arguments of set functions
- In the search condition of a WHERE clause

Although these functions are not defined by the ANSI SQL-92 standard, they are valuable because they are efficient, fast, and easy to use, and they simplify the expression of commonly asked business questions.

**Usage Notes**

Most of the RISQL display functions are order-dependent; that is, they operate on an ordered result set. Therefore, queries containing these functions typically contain an ORDER BY clause. RISQL display functions are calculated after the processing of the ORDER BY is complete.

Many of the queries in this chapter rely on aggregated sales totals. Because the Sales table stores only daily totals, it would be useful to create aggregate tables to answer these queries. If you are using the Red Brick Vista option to accelerate the performance of aggregate queries, refer to the Red Brick Vista User's Guide for instructions on creating and using aggregate tables.
**CUME()**

*Question*

What were the daily sales figures for Aroma Roma coffee during January, 1996? What were the cumulative subtotals for dollars and quantities during the month?

*Query*

```sql
select date, sum(dollars) as total_dollars,
    cume(sum(dollars)) as run_dollars,
    sum(quantity) as total_qty,
    cume(sum(quantity)) as run_qty
from period natural join sales
    natural join product
where year = 1996
    and month = 'JAN'
    and prod_name = 'Aroma Roma'
group by date
order by date;
```

*Result*

<table>
<thead>
<tr>
<th>DATE</th>
<th>TOTAL_DOLLARS</th>
<th>RUN_DOLLARS</th>
<th>TOTAL_QTY</th>
<th>RUN_QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996-01-02</td>
<td>855.50</td>
<td>855.50</td>
<td>118</td>
<td>118</td>
</tr>
<tr>
<td>1996-01-03</td>
<td>536.50</td>
<td>1392.00</td>
<td>74</td>
<td>192</td>
</tr>
<tr>
<td>1996-01-04</td>
<td>181.25</td>
<td>1573.25</td>
<td>25</td>
<td>217</td>
</tr>
<tr>
<td>1996-01-05</td>
<td>362.50</td>
<td>1935.75</td>
<td>50</td>
<td>267</td>
</tr>
<tr>
<td>1996-01-06</td>
<td>667.00</td>
<td>2602.75</td>
<td>92</td>
<td>359</td>
</tr>
<tr>
<td>1996-01-07</td>
<td>659.75</td>
<td>3262.50</td>
<td>91</td>
<td>450</td>
</tr>
<tr>
<td>1996-01-08</td>
<td>309.50</td>
<td>3572.00</td>
<td>54</td>
<td>504</td>
</tr>
<tr>
<td>1996-01-09</td>
<td>195.75</td>
<td>3767.75</td>
<td>27</td>
<td>531</td>
</tr>
<tr>
<td>1996-01-10</td>
<td>420.50</td>
<td>4188.25</td>
<td>58</td>
<td>589</td>
</tr>
<tr>
<td>1996-01-11</td>
<td>547.50</td>
<td>4735.75</td>
<td>78</td>
<td>667</td>
</tr>
<tr>
<td>1996-01-12</td>
<td>536.50</td>
<td>5272.25</td>
<td>74</td>
<td>741</td>
</tr>
<tr>
<td>1996-01-13</td>
<td>638.00</td>
<td>5910.25</td>
<td>88</td>
<td>829</td>
</tr>
<tr>
<td>1996-01-14</td>
<td>1057.50</td>
<td>6967.75</td>
<td>150</td>
<td>979</td>
</tr>
<tr>
<td>1996-01-15</td>
<td>884.50</td>
<td>7852.25</td>
<td>122</td>
<td>1101</td>
</tr>
<tr>
<td>1996-01-16</td>
<td>761.25</td>
<td>8613.50</td>
<td>105</td>
<td>1206</td>
</tr>
<tr>
<td>1996-01-17</td>
<td>455.50</td>
<td>9069.00</td>
<td>66</td>
<td>1272</td>
</tr>
<tr>
<td>1996-01-18</td>
<td>768.50</td>
<td>9837.50</td>
<td>106</td>
<td>1378</td>
</tr>
<tr>
<td>1996-01-19</td>
<td>746.75</td>
<td>10584.25</td>
<td>103</td>
<td>1481</td>
</tr>
<tr>
<td>1996-01-20</td>
<td>261.00</td>
<td>10845.25</td>
<td>36</td>
<td>1517</td>
</tr>
<tr>
<td>1996-01-21</td>
<td>630.75</td>
<td>11476.00</td>
<td>87</td>
<td>1604</td>
</tr>
<tr>
<td>1996-01-22</td>
<td>813.75</td>
<td>12289.75</td>
<td>115</td>
<td>1719</td>
</tr>
</tbody>
</table>

...
Cumulative Totals: **CUME**

The RISQL CUME function calculates and displays running totals. Although a standard SQL SELECT statement can calculate and display sales figures for a given period of time, it cannot calculate a cumulative total on a result set.

**CUME Function**

To calculate a cumulative total, place a CUME function in the select list for each numeric column to be summed:

```
CUME(expression)
```

where `expression` is a column name or a numeric expression.

**About the Example**

This query calculates the daily sales figures and quantities for Aroma Roma coffee during January of 1996. The query also calculates and displays cumulative totals for those values.

**Usage Notes**

The CUME function maintains a running total for a numeric expression that may, but does not have to, contain a column reference. For example:

```
select cume(1) as row_num, order_no, price from orders;
```

<table>
<thead>
<tr>
<th>ROW_NUM</th>
<th>ORDER_NO</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3600</td>
<td>1200.46</td>
</tr>
<tr>
<td>2</td>
<td>3601</td>
<td>1535.94</td>
</tr>
<tr>
<td>3</td>
<td>3602</td>
<td>780.00</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If a column reference occurs in the expression, it must be a numeric column.

When a query contains a GROUP BY clause, each expression in the select list must either reference one of the columns that occurs in the GROUP BY clause or be an SQL set function or RISQL display function.

An ORDER BY clause is included to ensure that the CUME function operates on an ordered set of rows. Columns used in ORDER BY clauses must exist in the GROUP BY clause; therefore, this result set is both grouped and ordered by the Date column.
Data Analysis

CUME with RESET BY

Question

What were the cumulative daily Aroma Roma sales figures during each week of January, 1996?

Query

```sql
select week, date, sum(dollars) as total_dollars,
    cume(sum(dollars)) as run_dollars,
    sum(quantity) as total_qty,
    cume(sum(quantity)) as run_qty
from period natural join sales
    natural join product
where year = 1996
    and month = 'JAN'
    and prod_name = 'Aroma Roma'
group by week, date
order by week, date
reset by week;
```

Result

<table>
<thead>
<tr>
<th>WEEK</th>
<th>DATE</th>
<th>TOTAL_DOLLARS</th>
<th>RUN_DOLLARS</th>
<th>TOTAL_QTY</th>
<th>RUN_QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1996-01-02</td>
<td>855.50</td>
<td>855.50</td>
<td>118</td>
<td>118</td>
</tr>
<tr>
<td>1</td>
<td>1996-01-03</td>
<td>536.50</td>
<td>1392.00</td>
<td>74</td>
<td>192</td>
</tr>
<tr>
<td>1</td>
<td>1996-01-04</td>
<td>181.25</td>
<td>1573.25</td>
<td>25</td>
<td>217</td>
</tr>
<tr>
<td>1</td>
<td>1996-01-05</td>
<td>362.50</td>
<td>1935.75</td>
<td>50</td>
<td>267</td>
</tr>
<tr>
<td>1</td>
<td>1996-01-06</td>
<td>667.00</td>
<td>2602.75</td>
<td>92</td>
<td>359</td>
</tr>
<tr>
<td>2</td>
<td>1996-01-07</td>
<td>659.75</td>
<td>659.75</td>
<td>91</td>
<td>91</td>
</tr>
<tr>
<td>2</td>
<td>1996-01-08</td>
<td>309.50</td>
<td>969.25</td>
<td>54</td>
<td>145</td>
</tr>
<tr>
<td>2</td>
<td>1996-01-09</td>
<td>195.75</td>
<td>1165.00</td>
<td>27</td>
<td>172</td>
</tr>
<tr>
<td>2</td>
<td>1996-01-10</td>
<td>420.50</td>
<td>1585.50</td>
<td>58</td>
<td>230</td>
</tr>
<tr>
<td>2</td>
<td>1996-01-11</td>
<td>547.50</td>
<td>2133.00</td>
<td>78</td>
<td>308</td>
</tr>
<tr>
<td>2</td>
<td>1996-01-12</td>
<td>536.50</td>
<td>2669.50</td>
<td>74</td>
<td>382</td>
</tr>
<tr>
<td>2</td>
<td>1996-01-13</td>
<td>638.00</td>
<td>3307.50</td>
<td>88</td>
<td>470</td>
</tr>
<tr>
<td>3</td>
<td>1996-01-14</td>
<td>1057.50</td>
<td>1057.50</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>3</td>
<td>1996-01-15</td>
<td>884.50</td>
<td>1942.00</td>
<td>122</td>
<td>272</td>
</tr>
<tr>
<td>3</td>
<td>1996-01-16</td>
<td>761.25</td>
<td>2703.25</td>
<td>105</td>
<td>377</td>
</tr>
<tr>
<td>3</td>
<td>1996-01-17</td>
<td>455.50</td>
<td>3158.75</td>
<td>66</td>
<td>443</td>
</tr>
<tr>
<td>3</td>
<td>1996-01-18</td>
<td>768.50</td>
<td>3927.25</td>
<td>106</td>
<td>549</td>
</tr>
<tr>
<td>3</td>
<td>1996-01-19</td>
<td>746.75</td>
<td>4674.00</td>
<td>103</td>
<td>652</td>
</tr>
<tr>
<td>3</td>
<td>1996-01-20</td>
<td>261.00</td>
<td>4935.00</td>
<td>36</td>
<td>688</td>
</tr>
<tr>
<td>4</td>
<td>1996-01-21</td>
<td>630.75</td>
<td>630.75</td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td>4</td>
<td>1996-01-22</td>
<td>813.75</td>
<td>1444.50</td>
<td>115</td>
<td>202</td>
</tr>
</tbody>
</table>

...
Resetting Cumulative Totals: RESET BY Clause

You can reset the calculation of cumulative subtotals for multiple columns by including a RESET BY subclause in the ORDER BY clause. This subclause resets running totals to zero whenever the values in the specified columns change. Client tools often refer to such changes as control breaks or programmed breaks.

RESET BY Subclause

This subclause must occur within an ORDER BY clause:

```
SELECT select_list
FROM table_list
[WHERE search_condition]
[GROUP BY group_list]
[HAVING search_condition]
[ORDER BY order_list
  [RESET BY reset_list]]
  [BREAK BY order_reference SUMMING select_reference_list]
  [SUPPRESS BY column_list];
```

where `reset_list` is one or more columns listed in the select list.

About the Example

This query calculates running totals for sales of Aroma Roma coffee during January, 1996. The RESET BY subclause resets the running total to zero when the Week value changes. To do this, the query must be ordered and grouped by the Week column; therefore, the Week column occurs in all three clauses—GROUP BY, ORDER BY, and RESET BY.

The blank line in the result set was produced by using the RISQL Reporter command SET COLUMN `column_name` SKIP LINE.

Usage Notes

Columns referenced in the RESET BY clause must occur in the select list and the ORDER BY clause. Positional references to columns in the select list may be used in the ORDER BY and RESET BY clauses but not in the GROUP BY clause.

For more information about the ORDER BY clause, refer to page 2-21.
**MOVINGAVG()**

**Question**
What was the three-week moving average of product sales at San Jose and Miami stores during the third quarter of 1995?

**Query**

```sql
select city, week, sum(dollars) as sales,
    string(movingavg(sum(dollars), 3), 7, 2) as mov_avg,
    cume(sum(dollars)) as run_sales
from store natural join sales natural join period
where qtr = 'Q3_95' and city in ('San Jose', 'Miami')
group by city, week
order by city, week
reset by city;
```

**Result**

<table>
<thead>
<tr>
<th>CITY</th>
<th>WEEK</th>
<th>SALES</th>
<th>MOV_AVG</th>
<th>RUN_SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miami</td>
<td>26</td>
<td>869.25</td>
<td>NULL</td>
<td>869.25</td>
</tr>
<tr>
<td>Miami</td>
<td>27</td>
<td>3727.45</td>
<td>NULL</td>
<td>4596.70</td>
</tr>
<tr>
<td>Miami</td>
<td>28</td>
<td>4850.20</td>
<td>3148.96</td>
<td>9464.90</td>
</tr>
<tr>
<td>Miami</td>
<td>29</td>
<td>4948.90</td>
<td>4508.85</td>
<td>14395.80</td>
</tr>
<tr>
<td>Miami</td>
<td>30</td>
<td>4385.65</td>
<td>4728.25</td>
<td>18781.45</td>
</tr>
<tr>
<td>Miami</td>
<td>31</td>
<td>3276.75</td>
<td>4203.76</td>
<td>22058.20</td>
</tr>
<tr>
<td>Miami</td>
<td>32</td>
<td>4131.75</td>
<td>3931.38</td>
<td>26189.95</td>
</tr>
<tr>
<td>Miami</td>
<td>33</td>
<td>3174.40</td>
<td>3527.63</td>
<td>29364.35</td>
</tr>
<tr>
<td>Miami</td>
<td>34</td>
<td>3781.75</td>
<td>3695.96</td>
<td>31461.10</td>
</tr>
<tr>
<td>Miami</td>
<td>35</td>
<td>4708.00</td>
<td>3888.05</td>
<td>37854.10</td>
</tr>
<tr>
<td>Miami</td>
<td>36</td>
<td>3868.55</td>
<td>4119.43</td>
<td>41722.65</td>
</tr>
<tr>
<td>Miami</td>
<td>37</td>
<td>6293.35</td>
<td>4956.63</td>
<td>48016.00</td>
</tr>
<tr>
<td>Miami</td>
<td>38</td>
<td>4998.75</td>
<td>5053.55</td>
<td>53014.75</td>
</tr>
<tr>
<td>Miami</td>
<td>39</td>
<td>5063.80</td>
<td>5451.96</td>
<td>58078.55</td>
</tr>
</tbody>
</table>

| San Jose | 26   | 2494.00 | NULL    | 2494.00    |
| San Jose | 27   | 4839.20 | NULL    | 7333.20    |
| San Jose | 28   | 7422.80 | 4918.66 | 14756.00   |
| San Jose | 29   | 8971.50 | 7077.83 | 23727.50   |
| San Jose | 30   | 6522.90 | 7639.06 | 30250.40   |
| San Jose | 31   | 6846.40 | 7446.93 | 37096.80   |
| San Jose | 32   | 7790.95 | 7053.41 | 44887.75   |
| San Jose | 33   | 6546.20 | 7061.18 | 51433.95   |
| San Jose | 34   | 7214.90 | 7184.01 | 58648.85   |
| San Jose | 35   | 6642.60 | 6801.23 | 65291.45   |
| San Jose | 36   | 8765.35 | 7540.95 | 74056.80   |
| San Jose | 37   | 7686.50 | 7698.15 | 81743.30   |
| San Jose | 38   | 8474.10 | 8308.65 | 90217.40   |
| San Jose | 39   | 7253.70 | 7804.76 | 97471.10   |
Calculating Moving Averages: MOVINGAVG

Sales figures fluctuate over time; when they fluctuate radically, they obscure underlying, long-range trends. Moving averages are used to smooth the effects of these fluctuations. For example, a three-week moving average divides the sum of the last three consecutive weekly aggregations by three.

MOVINGAVG Function

To calculate a moving average, place a MOVINGAVG function in the select list for each numeric column to be averaged. The function refers to a column or numeric expression to be averaged and an integer representing the number of rows to average:

\[
\text{MOVINGAVG}(n\_expression, n)
\]

where \( n\_expression \) is the name of a column containing numeric data or a numeric expression and \( n \) is an integer that represents a smoothing factor.

About the Example

This query calculates a three-week average for sales at San Jose and Miami stores during the third quarter of 1995. A control break is triggered by a RESET BY subclause on the City column. Three weeks must pass before a three-week moving average can be calculated; consequently, the first two rows following each control break contain NULLs.

The result table must be fully ordered, and the moving average must be reset when the city changes. If the rows are not in chronological order, the moving average function returns incorrect results. Therefore, the ORDER BY clause includes both the City column and the Week column.

The STRING scalar function is used to truncate the long-numeric values returned for the Mov_Avg column. For details, refer to the SQL Reference Guide.

As in the previous example, the blank line in the result set was produced by using the RISQL Reporter command SET COLUMN column_name SKIP LINE.

Usage Notes

ORDER BY clauses are recommended for all queries that contain order-dependent RISQL display functions.
**MOVING SUM**

**Question**

What was the seven-day moving sum of quantities of Demitasse Ms coffee sold during March, 1996?

**Query**

```sql
select date, sum(quantity) as day_qty,
       string(movingsum(sum(quantity), 7),7,2) as mov_sum
from store natural join sales natural join period
     natural join product
where year = 1996 and month = 'MAR' and prod_name = 'Demitasse Ms'
group by date
order by date;
```

**Result**

```
DATE    DAY_QTY  MOV_SUM
1996-03-01  65       NULL
1996-03-02  19       NULL
1996-03-03  92       NULL
1996-03-04  91       NULL
1996-03-05 106       NULL
1996-03-06  92       NULL
1996-03-07 102      567.00
1996-03-08  21      523.00
1996-03-09  74      578.00
1996-03-10  81      567.00
1996-03-11  77      553.00
1996-03-12 127      574.00
1996-03-13 169      651.00
1996-03-14  31      580.00
1996-03-15  56      615.00
1996-03-16  40      581.00
1996-03-17  84      584.00
1996-03-18  34      541.00
1996-03-19 128      542.00
1996-03-20  97      470.00
1996-03-21  50      489.00
1996-03-22 147      580.00
1996-03-23 104      644.00
1996-03-24  48      608.00
1996-03-25  93      667.00
1996-03-26 130      669.00
1996-03-27  95      667.00
1996-03-28 122      739.00
...```

3-10   SQL Self-Study Guide
Calculating Moving Sums: MOVINGSUM

A moving sum function, like a moving average, is used to smooth the effects of fluctuations. For example, a seven-day moving sum is calculated by summing seven consecutive days.

**MOVINGSUM Function**

To calculate a moving sum, place a MOVINGSUM function in the select list for each numeric column to be summed. The function requires a column name or numeric expression (`n_expression`) indicating the column to be summed and an integer (`n`) representing the number of rows to sum (a smoothing factor):

```
MOVINGSUM(n_expression, n)
```

**About the Example**

This query calculates a seven-day moving sum of quantities of Demitasse Ms coffee sold during March, 1996. The first six rows have NULL entries because seven days must pass before the moving sum can be calculated.

The STRING scalar function is used to truncate the long-numeric values returned for the Mov_Sum column. For more information about this function, refer to the *SQL Reference Guide*.

**Usage Notes**

If the MOVINGSUM function is applied to results that are not in chronological order, the function will return incorrect results; therefore, an ORDER BY clause is recommended. As in the previous example, the Date column must be used in both the ORDER BY clause and the GROUP BY clause.
**RANK**

**Question**

What were the March 1995 rankings of stores in the Western region, in terms of total dollar sales?

**Query**

```sql
select store_name, district, sum(dollars) as total_sales, 
    rank(sum(dollars)) as sales_rank 
from market natural join store 
    natural join sales 
    natural join period 
where year = 1995 
    and month = 'MAR' 
    and region = 'West' 
group by store_name, district;
```

**Result**

<table>
<thead>
<tr>
<th>STORE_NAME</th>
<th>DISTRICT</th>
<th>TOTAL_SALES</th>
<th>SALES_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cupertino Coffee Supply</td>
<td>San Francisco</td>
<td>18801.50</td>
<td>1</td>
</tr>
<tr>
<td>San Jose Roasting Company</td>
<td>San Francisco</td>
<td>18346.90</td>
<td>2</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Los Angeles</td>
<td>18282.05</td>
<td>3</td>
</tr>
<tr>
<td>Java Judy's</td>
<td>Los Angeles</td>
<td>17826.25</td>
<td>4</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>San Francisco</td>
<td>15650.50</td>
<td>5</td>
</tr>
<tr>
<td>Roasters, Los Gatos</td>
<td>San Francisco</td>
<td>12694.50</td>
<td>6</td>
</tr>
</tbody>
</table>
**Ranking Data: RANK**

You can rank any set of values with the RANK function, which assigns 1 to the largest value in a group, 2 to the next largest, and so forth. Magnitude, not order, determines the rank of a value.

**RANK Function**

To rank a set of values, specify

```sql
RANK(expression)
```

in the select list, where `expression` is a column name or a numeric expression. If `expression` is NULL, RANK returns NULL. For more information about numeric expressions, refer to the SQL Reference Guide.

**About the Example**

This example ranks stores in the Western region in terms of dollar sales for March 1995. The daily totals from the Sales table that meet the search conditions in the WHERE clause are summed, then ranked.

**Usage Notes**

The GROUP BY clause is required in this example. In a SELECT statement without a GROUP BY clause, where an aggregation function is included in the select list, all column references must be aggregation functions.

RANK is not an order-dependent display function; by default, queries that contain a RANK function and no ORDER BY clause sort the result set by the ranking values (highest to lowest).

To rank a set of values from bottom to top, reverse the sign of the ranked column with the unary negation operator:

```sql
RANK(-expression)
```

where `expression` is a column name or a numeric expression. For example:

```sql
rank(-dollars) as sales_rank
```
**RANK, WHEN**

**Question**

In the first quarter of 1996 at the Olympic Coffee Company, what were the top 10 days for sales of Breakfast Blend tea? What were the corresponding ranks by quantity?

**Query**

```sql
select date, day, dollars as day_sales,
    rank(dollars) as sales_rank,
    quantity as day_qty,
    rank(quantity) as qty_rank
from product natural join sales
    natural join period
    natural join store
where qtr = 'Q1_96'
    and prod_name like 'Break%'
    and store_name like 'Olympic%'
when sales_rank <= 10
order by date;
```

**Result**

<table>
<thead>
<tr>
<th>DATE</th>
<th>DAY</th>
<th>DAY_SALES</th>
<th>SALES_RANK</th>
<th>DAY_QTY</th>
<th>QTY_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996-01-21</td>
<td>SU</td>
<td>30.00</td>
<td>9</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1996-02-01</td>
<td>TH</td>
<td>56.25</td>
<td>3</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>1996-02-08</td>
<td>TH</td>
<td>30.00</td>
<td>9</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1996-02-22</td>
<td>TH</td>
<td>71.25</td>
<td>1</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>1996-02-23</td>
<td>FR</td>
<td>41.25</td>
<td>7</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>1996-03-03</td>
<td>SU</td>
<td>59.50</td>
<td>2</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>1996-03-11</td>
<td>MO</td>
<td>55.25</td>
<td>5</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>1996-03-16</td>
<td>SA</td>
<td>56.25</td>
<td>3</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>1996-03-22</td>
<td>FR</td>
<td>38.25</td>
<td>8</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>1996-03-23</td>
<td>SA</td>
<td>42.50</td>
<td>6</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>
Ranking the Top Ten: RANK, WHEN

You can rank any set of numeric values with the RANK function and then specify that only those of interest are displayed. For example, you can rank the sales of all products but display only the top ten, the bottom ten, or any other combination that can be expressed in a search condition.

WHEN Clause

To restrict the rows returned in a result table after ranked values have been calculated (or other display functions or set functions), include a WHEN clause in the query.

```
SELECT select_list
FROM table_name
[WHERE search_condition]
[GROUP BY group_list]
[HAVING search_condition]
[WHEN condition]
[ORDER BY order_list]
[RESET BY reset_list]
[BREAK BY order_reference SUMMING select_reference_list]
[SUPPRESS BY column_list];
```

Compound conditions constructed with the AND, OR, and NOT logical connectives are allowed in the WHEN clause. For more information about conditions, refer to Chapter 3 of the SQL Reference Guide.

About the Example

This query ranks daily sales of Breakfast Blend tea at a single store in a single quarter, but returns figures only for the top 10 days. The query also returns the corresponding quantity rankings. The ORDER BY clause sorts the result set in chronological order (by the values in the Date column).

When values to be ranked “tie,” they are assigned the same ranking value. For example, two rows receive a sales rank of 3 in this case.
Data Analysis

**NTILE**

**Question**

Which products rank in the top 25% and bottom 25% based on annual sales totals for 1995?

**Query**

```sql
select prod_name, sum(dollars) as total_sales,
       ntile(total_sales, 4) as sales_rank
from sales natural join product
     natural join period
where year = 1995
group by prod_name
when sales_rank in (1, 4);
```

**Result**

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>TOTAL_SALES</th>
<th>SALES_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demitasse Ms</td>
<td>304727.00</td>
<td>1</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>263353.00</td>
<td>1</td>
</tr>
<tr>
<td>NA Lite</td>
<td>262162.00</td>
<td>1</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>251713.00</td>
<td>1</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>251086.50</td>
<td>1</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>229201.25</td>
<td>1</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>227769.50</td>
<td>1</td>
</tr>
<tr>
<td>La Antigua</td>
<td>223528.25</td>
<td>1</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>218574.75</td>
<td>1</td>
</tr>
<tr>
<td>Colombiano</td>
<td>218009.50</td>
<td>1</td>
</tr>
<tr>
<td>Aroma Sounds CD</td>
<td>5937.00</td>
<td>4</td>
</tr>
<tr>
<td>Aroma Sounds Cassette</td>
<td>5323.00</td>
<td>4</td>
</tr>
<tr>
<td>French Press, 4-Cup</td>
<td>4570.50</td>
<td>4</td>
</tr>
<tr>
<td>Spice Jar</td>
<td>4073.00</td>
<td>4</td>
</tr>
<tr>
<td>French Press, 2-Cup</td>
<td>3042.75</td>
<td>4</td>
</tr>
<tr>
<td>Travel Mug</td>
<td>1581.75</td>
<td>4</td>
</tr>
<tr>
<td>Easter Sampler Basket</td>
<td>1500.00</td>
<td>4</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>1258.00</td>
<td>4</td>
</tr>
<tr>
<td>Christmas Sampler</td>
<td>1230.00</td>
<td>4</td>
</tr>
</tbody>
</table>
Ranking Values in Groups: NTILE

You can rank each value in a group of numeric values as 1 (highest) through any specified number (lowest) by using the NTILE function. This function assigns the appropriate rank to a value depending on its magnitude relative to other values in a group.

Syntax

To rank a set of values into 100 equal groups, include the NTILE function in the select list, specify the numeric expression or column to be ranked, followed by the number 100.

\[
\text{NTILE(expression, 100)}
\]

If \( expression \) is NULL, NTILE returns NULL. For more information about numeric expressions, refer to the SQL Reference Guide.

About the Example

This example query ranks products as 1, 2, 3, or 4, based on annual sales totals for 1995. The WHEN clause removes the middle 50% (2 and 3) from the result set.

Usage Notes

In those cases where equal values span a boundary, they are distributed between adjacent groups.

If the set of values is not divisible by the specified number, the NTILE function puts leftover rows in the higher-level group.

Tip: By using the NTILE function inside a CASE expression, you can redistribute ranked values into unequal groups and replace the default NTILE numeric values with more meaningful labels. For an example, refer to “NTILE with CASE” on page 3-18.
**Data Analysis**

**NTILE with CASE**

**Question**

What products fell into the top 20%, middle 60%, and bottom 20% of sales totals for the second week of 1996, at stores in the Western region?

**Query**

```sql
select prod_name, sum(quantity) as quantity, sum(dollars) as sales, case ntile(sum(dollars), 5)
when 1 then 'TOP_20'
when 2 then 'MID_60'
when 3 then 'MID_60'
when 4 then 'MID_60'
when 5 then 'LOW_20'
end as grp
from market natural join store natural join sales natural join period natural join product
where year = 1996 and week = 2 and region = 'West'
group by prod_name;
```

**Result**

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>QUANTITY</th>
<th>SALES</th>
<th>GRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA Lite</td>
<td>281</td>
<td>2529.00</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>256</td>
<td>2067.50</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>280</td>
<td>1948.00</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Colombiano</td>
<td>215</td>
<td>1441.75</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>162</td>
<td>1363.50</td>
<td>TOP_20</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>137</td>
<td>1352.75</td>
<td>TOP_20</td>
</tr>
<tr>
<td>La Antigua</td>
<td>182</td>
<td>1314.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>161</td>
<td>1260.75</td>
<td>MID_60</td>
</tr>
<tr>
<td>Ruby's Allspice</td>
<td>210</td>
<td>1195.00</td>
<td>MID_60</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>100</td>
<td>1167.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>114</td>
<td>945.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>204</td>
<td>850.00</td>
<td>MID_60</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>108</td>
<td>824.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>91</td>
<td>564.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>97</td>
<td>547.75</td>
<td>MID_60</td>
</tr>
<tr>
<td>Special Tips</td>
<td>139</td>
<td>476.75</td>
<td>MID_60</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>99</td>
<td>412.25</td>
<td>MID_60</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>35</td>
<td>383.25</td>
<td>MID_60</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>81</td>
<td>313.75</td>
<td>MID_60</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>79</td>
<td>287.50</td>
<td>MID_60</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>81</td>
<td>279.75</td>
<td>MID_60</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>71</td>
<td>244.00</td>
<td>MID_60</td>
</tr>
<tr>
<td>Espresso Machine Italiano</td>
<td>2</td>
<td>199.90</td>
<td>MID_60</td>
</tr>
<tr>
<td>Aroma Sounds CD</td>
<td>10</td>
<td>130.00</td>
<td>LOW_20</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>10</td>
<td>79.50</td>
<td>LOW_20</td>
</tr>
<tr>
<td>Spice Jar</td>
<td>4</td>
<td>72.00</td>
<td>LOW_20</td>
</tr>
<tr>
<td>Spice Sampler</td>
<td>4</td>
<td>48.00</td>
<td>LOW_20</td>
</tr>
<tr>
<td>Coffee Mug</td>
<td>6</td>
<td>30.00</td>
<td>LOW_20</td>
</tr>
</tbody>
</table>
Ranking Values in Unequal Groups: CASE and NTILE

The NTILE function can combine powerfully with a CASE expression to rank and redistribute a set of values. For example, NTILE might be used to rank values into five equal groups; then a CASE expression could be used to redistribute those values into unequal groups, representing a curve.

**CASE Syntax**

A CASE expression is a conditional scalar expression that can be used in the select list to substitute a specified column value for another value:

```
CASE expression WHEN result THEN result1 ELSE result2 END AS col_alias
```

where `expression` is any valid expression; `result` is a value to which the expression is expected to evaluate; and `result1` is a value that substitutes for `result`. When `expression` does not evaluate to `result`, a default value (`result2`) is used, if specified. Typically, multiple WHEN...THEN conditions are used to find and replace several different values.

**Note:** A CASE expression can take one of two forms: *simple* or *searched*. This example uses the simple form. For more details, refer to the SQL Reference Guide. For an example of the searched form, see page 4-5.

**About the Example**

The example on the facing page uses the NTILE function inside a CASE expression to spread the tiled values (fifths) into three unequal groups: the top 20% are represented in the final result set as TOP_20, the middle 60% as MID_60, and the bottom 20% as LOW_20. When the expression `ntile(sum(dollars), 5)` evaluates to 1, that value is substituted with the character string TOP_20. When the expression evaluates to 2, 3, or 4, those values are replaced with MID_60, and when it evaluates to 5, it is replaced with LOW_20.

**Usage Notes**

By adding a WHEN clause to the end of this query, you could eliminate a specified portion of the result set. For example:

```
when grp = 'MID_60'
```
**Data Analysis**

**TERTILE**

**Question**

Which cities ranked high, middle, and low in the West and South in 1995, in terms of unit sales of Earl Grey tea?

**Query**

```sql
select city, sum(quantity) as qty_1995,
       tertile(sum(quantity)) as q_rk
from market
    natural join store
    natural join sales
    natural join product
    natural join period
where year = 1995
    and prod_name like 'Earl Grey%' 
    and region in ('West', 'South')
group by city;
```

**Result**

<table>
<thead>
<tr>
<th>CITY</th>
<th>QTY_1995</th>
<th>Q_RK</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Jose</td>
<td>1469</td>
<td>H</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>911</td>
<td>H</td>
</tr>
<tr>
<td>Phoenix</td>
<td>814</td>
<td>H</td>
</tr>
<tr>
<td>Los Gatos</td>
<td>805</td>
<td>M</td>
</tr>
<tr>
<td>Miami</td>
<td>782</td>
<td>M</td>
</tr>
<tr>
<td>Cupertino</td>
<td>778</td>
<td>M</td>
</tr>
<tr>
<td>Houston</td>
<td>768</td>
<td>L</td>
</tr>
<tr>
<td>New Orleans</td>
<td>684</td>
<td>L</td>
</tr>
<tr>
<td>Atlanta</td>
<td>614</td>
<td>L</td>
</tr>
</tbody>
</table>
Ranking Values as High, Middle, or Low: TERTILE

You can rank each value in a group of numeric values as High, Middle, or Low with the TERTILE function. This function assigns the letter H, M, or L to a value depending on its magnitude relative to other values in a group.

Syntax

To rank a group into thirds, include the TERTILE function in the select list and specify the numeric expression or column to be ranked.

\[
\text{TERTILE(expression)}
\]

where \( \text{expression} \) is a column name or a numeric expression. If \( \text{expression} \) is NULL, TERTILE returns NULL. For more information about numeric expressions, refer to the SQL Reference Guide.

About the Example

This example query ranks cities according to quantities of Earl Grey tea sold during 1995. The result table is divided into city groups and the Quantity column is summed for each city for the year.

Usage Notes

Values do not always fall nicely into three sets. When a set of values is not divisible by three, the TERTILE function puts any leftover rows in the higher-level group. In those cases where equal values span a boundary, they are distributed between adjacent groups.

Even though the column referenced by the TERTILE function must be numeric, the result of this function is always a character column.

For more information about the TERTILE function and datatypes, refer to the SQL Reference Guide.
**RATIO TO REPORT**

**Question**

What was the ratio of monthly sales to total sales of Xalapa Lapa coffee in San Jose and Los Angeles stores during the third quarter of 1995?

**Query**

```sql
select city, month, sum(dollars) as total_sales, 
  ratiototreport(sum(dollars))*100 as pct_of_sales 
from store natural join sales 
  natural join product 
  natural join period 
where prod_name like 'Xalapa%' 
  and qtr = 'Q3_95' 
  and city in ('San Jose', 'Los Angeles') 
group by city, month;
```

**Result**

<table>
<thead>
<tr>
<th>CITY</th>
<th>MONTH</th>
<th>TOTAL_SALES</th>
<th>PCT_OF_SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Jose</td>
<td>JUL</td>
<td>2499.50</td>
<td>26.99</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>JUL</td>
<td>1627.00</td>
<td>17.57</td>
</tr>
<tr>
<td>San Jose</td>
<td>AUG</td>
<td>1004.00</td>
<td>10.84</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>AUG</td>
<td>995.00</td>
<td>10.74</td>
</tr>
<tr>
<td>San Jose</td>
<td>SEP</td>
<td>1802.00</td>
<td>19.46</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>SEP</td>
<td>1334.00</td>
<td>14.40</td>
</tr>
</tbody>
</table>
Calculating Ratios as Percentages: RATIO TO REPORT *100

The RATIO TO REPORT function calculates the ratio of a numeric row value to the total value of that column in the result set. For example, if a given column lists sales figures for different products, each value in that column can be expressed as a ratio of the total sales for all the products listed.

Syntax

To calculate a ratio of a column value to the sum of all the values in the column, include the RATIO TO REPORT function in the select list and specify a numeric expression or the name of a column that contains numeric values.

\[ \text{RATIO TO REPORT(} \text{expression}) \]

If \( \text{expression} \) is NULL, RATIO TO REPORT returns NULL. For more information about numeric expressions, refer to the \textit{SQL Reference Guide}.

To calculate ratios as percentages, simply use the notation \( \times 100 \) after the expression.

About the Example

This example displays the ratio of monthly sales of Xalapa Lapa coffee in San Jose and Los Angeles stores during the third quarter of 1995 to the total sales of that product in those stores during the same period.

The expression

\[ (\text{sum(dollars)}) \times 100 \]

returns the results of the RATIO TO REPORT function as percentages. (The values in the Pct_of_Sales column add up to exactly 100.)

Usage Notes

The RATIO TO REPORT function can be reset for groups of values with the RESET BY subclause of the ORDER BY clause. For information about RESET BY, refer to page 3-7.
**Data Analysis**

**DATEADD**

**Question**

Calculate a date 90 days prior to and 90 days after a given date.

**Query**

```sql
select dateadd(day, -90, date) as due_date,
       date as cur_date,
       dateadd(day, 90, date) as past_due
from period
where year = 1996
      and month = 'JAN';
```

**Result**

<table>
<thead>
<tr>
<th>DUE_DATE</th>
<th>CUR_DATE</th>
<th>PAST_DUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995-10-03</td>
<td>1996-01-01</td>
<td>1996-03-31</td>
</tr>
<tr>
<td>1995-10-05</td>
<td>1996-01-03</td>
<td>1996-04-02</td>
</tr>
<tr>
<td>1995-10-06</td>
<td>1996-01-04</td>
<td>1996-04-03</td>
</tr>
<tr>
<td>1995-10-08</td>
<td>1996-01-06</td>
<td>1996-04-05</td>
</tr>
<tr>
<td>1995-10-09</td>
<td>1996-01-07</td>
<td>1996-04-06</td>
</tr>
<tr>
<td>1995-10-10</td>
<td>1996-01-08</td>
<td>1996-04-07</td>
</tr>
<tr>
<td>1995-10-11</td>
<td>1996-01-09</td>
<td>1996-04-08</td>
</tr>
<tr>
<td>1995-10-12</td>
<td>1996-01-10</td>
<td>1996-04-09</td>
</tr>
<tr>
<td>1995-10-20</td>
<td>1996-01-18</td>
<td>1996-04-17</td>
</tr>
</tbody>
</table>
Incrementing or Decrementing Dates: DATEADD

The DATEADD function returns a datetime value calculated from three arguments:

- Datepart that specifies an increment measure such as day, month, or year.
- Positive or negative increment value.
- Value to be incremented or decremented (column name or datetime expression).

### Function Returns

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEADD(day, 90, '07-01-95')</td>
<td>1995-09-29</td>
</tr>
<tr>
<td>DATEADD(month, 3, '07-01-95')</td>
<td>1995-10-01</td>
</tr>
<tr>
<td>DATEADD(year, 1, '07-01-95')</td>
<td>1996-07-01</td>
</tr>
</tbody>
</table>

**About the Example**

The example on the facing page calculates a date 90 days before and 90 days after a given date. The DATEADD function returns the value in the ANSI SQL-92 datetime format.

You can also reformat the DATETIME value as a month name using the DATENAME function. The following query uses the DATENAME function in its WHERE clause.

```sql
SELECT datename(month, dateadd(day, -90, date)) AS prior,
       datename(month, date) AS cur,
       datename(month, dateadd(day, 90, date)) AS next
FROM period
WHERE datename(yy, date) = '1996'
  AND month = 'JAN';
```

<table>
<thead>
<tr>
<th>PRIOR</th>
<th>CUR</th>
<th>NEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>October</td>
<td>January</td>
<td>March</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
<tr>
<td>October</td>
<td>January</td>
<td>April</td>
</tr>
</tbody>
</table>

... 

Because there were 29 days in February of 1996, January 1 plus 90 days is March 31; therefore, the first row in the Next column is March. For more information about DATETIME functions, refer to the SQL Reference Guide.
**Data Analysis**

**DATEDIFF**

**Question**

How long did the storewide Christmas special promotion run in 1995?

**Query**

```sql
select promo_desc, year,
       datediff(day, end_date, start_date)+1 as days_on_promo
from promotion p, period d
where p.start_date = d.date
  and promo_desc like 'Christmas%'
  and year = 1995;
```

**Result**

<table>
<thead>
<tr>
<th>PROMO_DESC</th>
<th>YEAR</th>
<th>DAYS_ON_PROMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christmas special</td>
<td>1995</td>
<td>31</td>
</tr>
</tbody>
</table>
Calculating Elapsed Days: DATEDIFF

The DATEDIFF function returns a datetime value calculated from three arguments:

- Datepart that specifies the increment measure such as day, month, or year.
- Two datetime expressions, which must be DATE, TIME, or TIMESTAMP datatypes.

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEDIFF(day, '07-01-96','01-01-96')</td>
<td>182</td>
</tr>
<tr>
<td>DATEDIFF(month, '07-01-96','01-01-96')</td>
<td>6</td>
</tr>
<tr>
<td>DATEDIFF(quarter, '07-01-96','01-01-96')</td>
<td>2</td>
</tr>
</tbody>
</table>

About the Example

This example calculates the number of days elapsed between the start and finish of a storewide promotion. The DATEDIFF function

\[
datediff(day, \text{end\_date}, \text{start\_date}) + 1
\]

operates on the datetime values in the Promotion table to return the result.

The +1 is required because the difference between the End_Date and Start_Date values is equal to 30 days, whereas the duration of the promotion includes both the Start_Date and the End_Date (31 days).

Usage Notes

This query is also an example of a join between two tables that do not have a primary key–foreign key relationship; the joining columns simply have comparable datetime datatypes:

\[
\text{where p.start\_date} = \text{d.date}
\]

Any two tables, including system tables, can be joined over comparable columns.

Note that the purpose of the join in this case is simply to return the Year value from the Period table; alternatively, this value could be extracted from the datetime columns in the Promotion table.
**Data Analysis**

**EXTRACT**

**Question**

What were the day and month names and numbers for the first six weeks of 1994 (as extracted from the datetime values in the Period table)?

**Query**

```sql
SELECT DATENAME(weekday, date) AS day_name,
       EXTRACT(weekday FROM date) AS day_num,
       EXTRACT(day FROM date) AS day,
       EXTRACT(dayofyear FROM date) AS day_yr,
       DATENAME(month, date) AS mo_name,
       EXTRACT(month FROM date) AS mo_num
FROM period
WHERE EXTRACT(year FROM date) = 1994
  AND EXTRACT(week FROM date) < 7;
```

**Result**

<table>
<thead>
<tr>
<th>DAY_NAME</th>
<th>DAY_NUM</th>
<th>DAY</th>
<th>DAY_YR</th>
<th>MO_NAME</th>
<th>MO_NUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saturday</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Wednesday</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Thursday</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>11</td>
<td>11</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Wednesday</td>
<td>4</td>
<td>12</td>
<td>12</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Thursday</td>
<td>5</td>
<td>13</td>
<td>13</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>14</td>
<td>14</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>15</td>
<td>15</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>16</td>
<td>16</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>17</td>
<td>17</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>18</td>
<td>18</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Wednesday</td>
<td>4</td>
<td>19</td>
<td>19</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Thursday</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>21</td>
<td>21</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>22</td>
<td>22</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>23</td>
<td>23</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>24</td>
<td>24</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>25</td>
<td>25</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Wednesday</td>
<td>4</td>
<td>26</td>
<td>26</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Thursday</td>
<td>5</td>
<td>27</td>
<td>27</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>28</td>
<td>28</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>29</td>
<td>29</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Sunday</td>
<td>1</td>
<td>30</td>
<td>30</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Monday</td>
<td>2</td>
<td>31</td>
<td>31</td>
<td>January</td>
<td>1</td>
</tr>
<tr>
<td>Tuesday</td>
<td>3</td>
<td>32</td>
<td></td>
<td>February</td>
<td>2</td>
</tr>
<tr>
<td>Wednesday</td>
<td>4</td>
<td>33</td>
<td></td>
<td>February</td>
<td>2</td>
</tr>
<tr>
<td>Thursday</td>
<td>5</td>
<td>34</td>
<td></td>
<td>February</td>
<td>2</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>35</td>
<td></td>
<td>February</td>
<td>2</td>
</tr>
<tr>
<td>Saturday</td>
<td>7</td>
<td>36</td>
<td></td>
<td>February</td>
<td>2</td>
</tr>
</tbody>
</table>
Displaying Dateparts as Integers: EXTRACT

The EXTRACT function returns an integer value representing a part of a DATETIME value. The function requires two arguments:

- Datepart that specifies the increment measure such as day, month, or year.
- Datetime expression (column name or DATETIME expression).

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract(weekday from date_col)</td>
<td>Weekday as an integer value from the set (1, 2, ..., 7)</td>
</tr>
<tr>
<td>extract(day from date_col)</td>
<td>Day of month as an integer from the set (1, 2, ..., 31)</td>
</tr>
</tbody>
</table>

About the Example

This query uses the DATENAME and EXTRACT scalar functions to return day and month names and day and month numbers for the first six weeks of 1994.

Except for the first week of the year, weeks typically begin on Sunday or Monday, depending on the territory specified in the warehouse locale. For information about locale specifications, refer to the Warehouse Administrator’s Guide and the Installation and Configuration Guide.
Data Analysis

Summary

**RISQL Display Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUME(expression)</td>
<td>Cumulative sum</td>
</tr>
<tr>
<td>MOVINGAVG(expression, n)</td>
<td>Average of the previous n rows</td>
</tr>
<tr>
<td>MOVINGSUM(expression, n)</td>
<td>Sum of the previous n rows</td>
</tr>
<tr>
<td>NTILE(expression, n)</td>
<td>n-level rank of values</td>
</tr>
<tr>
<td>RANK(expression)</td>
<td>Numeric rank of values</td>
</tr>
<tr>
<td>TERTILE(expression)</td>
<td>Three-level (high, medium, and low) rank of values</td>
</tr>
<tr>
<td>RATIOTOREPORT(expression)</td>
<td>Ratio of portion to total</td>
</tr>
</tbody>
</table>

The expression argument must be a numeric expression or a numeric column name.

**CASE Expressions**

CASE expressions in the select list are useful for substituting column values with other specified values, such as meaningful character strings to replace numeric values returned by display functions.

Another use of the CASE expression is discussed on page 4-5.

**DATETIME Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEADD</td>
<td>Adds interval to datetime value.</td>
</tr>
<tr>
<td>DATEDIFF</td>
<td>Subtracts the difference between two datetime values.</td>
</tr>
<tr>
<td>DATENAME</td>
<td>Extracts datepart component from datetime value as character string.</td>
</tr>
<tr>
<td>EXTRACT</td>
<td>Extracts datepart from datetime value as integer.</td>
</tr>
</tbody>
</table>
**Data Analysis**

**Summary**

This chapter described how to:

- Use RISQL display functions to perform data analysis, such as calculating ranks, moving averages, and cumulative sums.
- Redistribute ranked values into unequal groups and give meaningful labels to those values by using an NTILE calculation inside a CASE expression.
- Use DATETIME scalar functions to calculate and extract date information from DATETIME columns.
Chapter 4 focuses on queries that compare data. The chapter begins by illustrating the problem that confronts the query writer: how to use SQL to return a spreadsheet or “cross-tab” report rather than a standard, vertically ordered result set that is hard to read. The problem is solved by using either CASE expressions or subqueries.

The CASE solution is presented first, as a simple and concise way of comparing like groups of values. Then several examples of FROM clause and select-list subqueries are presented. These subqueries have the added value of being able to both compare data from different groups and include calculations against the compared values, such as share percentages over given time periods.

This chapter ends with a section on subqueries stated as conditions in the WHERE clause, which are useful for simpler comparison queries. This section also describes the ALL, SOME, ANY, and EXISTS predicates, which can be used to express conditions on subquery results.
**Comparison Queries**

**Comparing Data with SQL**

**Question**

How did sales of packaged coffee compare at stores in the western region in 1994?

**Query**

```sql
select store_name, prod_name, sum(dollars) as sales
from market natural join store
    natural join sales
    natural join period
    natural join product
    natural join class
where region like 'West%'
    and year = 1994
    and class_type = 'Pkg_coffee'
group by store_name, prod_name
order by store_name, prod_name;
```

**Result**

<table>
<thead>
<tr>
<th>STORE_NAME</th>
<th>PROD_NAME</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beaches Brew</td>
<td>Aroma Roma</td>
<td>3483.50</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Cafe Au Lait</td>
<td>3129.50</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Colombiano</td>
<td>2298.25</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Demitasse Ms</td>
<td>4529.25</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Expresso XO</td>
<td>4132.75</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>La Antigua</td>
<td>4219.75</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Lotta Latte</td>
<td>3468.00</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>NA Lite</td>
<td>4771.00</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Veracruzano</td>
<td>4443.00</td>
</tr>
<tr>
<td>Beaches Brew</td>
<td>Xalapa Lapa</td>
<td>4304.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Aroma Roma</td>
<td>4491.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Cafe Au Lait</td>
<td>4375.50</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Colombiano</td>
<td>2653.50</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Demitasse Ms</td>
<td>3936.50</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Expresso XO</td>
<td>4689.25</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>La Antigua</td>
<td>2932.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Lotta Latte</td>
<td>5146.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>NA Lite</td>
<td>4026.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Veracruzano</td>
<td>3285.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Xalapa Lapa</td>
<td>5784.00</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Aroma Roma</td>
<td>3485.25</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Cafe Au Lait</td>
<td>3599.50</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Colombiano</td>
<td>3321.75</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Demitasse Ms</td>
<td>5422.25</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Expresso XO</td>
<td>2851.00</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>La Antigua</td>
<td>2937.25</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Lotta Latte</td>
<td>4783.50</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>NA Lite</td>
<td>3740.00</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Veracruzano</td>
<td>4712.00</td>
</tr>
<tr>
<td>Instant Coffee</td>
<td>Xalapa Lapa</td>
<td>3698.00</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison Queries

A Simple Comparison Query

You can list the sales of a group of products at specific stores using a simple SELECT statement, but the format of the result table makes the values difficult to compare. For example, the partial result set on the facing page shows that La Antigua coffee was sold at several stores in the western region, but these figures are hard to isolate.

This kind of data is much easier to compare when it is formatted like a spreadsheet. There are two ways to produce a spreadsheet, or “cross-tab,” report: by using CASE expressions or subqueries. The following examples in this chapter illustrate both methods of writing comparison queries.

About the Example

This example returns 1994 sales figures for packaged coffee products sold at each store in the western region, but the format of the output data makes it difficult to compare the figures product by product, store by store.
Comparison Queries

CASE Expressions

Question

How did sales of packaged coffee compare at stores in the western region in 1994?

Query

```
select prod_name,
    sum(case when store_name = 'Beaches Brew' then dollars else 0 end) as Beaches,
    sum(case when store_name = 'Cupertino Coffee Supply' then dollars else 0 end) as Cupertino,
    sum(case when store_name = 'Roasters, Los Gatos' then dollars else 0 end) as RoastLG,
    sum(case when store_name = 'San Jose Roasting Company' then dollars else 0 end) as SJRoastCo,
    sum(case when store_name = 'Java Judy''s' then dollars else 0 end) as JavaJudy,
    sum(case when store_name = 'Instant Coffee' then dollars else 0 end) as Instant
from market natural join store
    natural join sales
    natural join product
    natural join class
where region like 'West%' and year = 1994
    and class_type = 'Pkg_coffee'
group by prod_name
order by prod_name;
```

Result

```
PROD_NAME      BEACHES  CUPERTINO  ROASTLG  SJROASTCO  JAVAJUDY  INSTANT
Aroma Roma     3483.50  4491.00  4602.00  4399.25  3748.25  3485.25
Cafe Au Lait   3129.50  4375.50  4199.00  3620.00  4864.50  3599.50
Colombiano     2298.25  2653.50  4205.00  3530.75  3509.00  3321.75
Demitasse Ms   4529.25  3936.50  4347.75  5699.00  6395.25  5422.25
Expresso XO    4132.75  4689.25  4234.50  3811.00  5012.25  2851.00
La Antigua     4219.75  2932.00  3447.50  4323.00  2410.25  2937.25
Lotta Latte    3468.00  5146.00  4469.50  5103.50  4003.00  4783.50
NA Lite        4771.00  4026.00  2736.00  4791.00  3740.00  3740.00
Veracruzano    4443.00  3285.00  4467.00  3856.00  4510.00  4712.00
Xalapa Lapa    4304.00  5784.00  3906.00  3645.00  3182.00  3698.00
```
A Solution for Comparing Data: CASE Expressions

An efficient and concise way to display compared values in a readable spreadsheet format is to use CASE expressions in the select list. Each CASE operation evaluates a specified expression and supplies a different value depending on whether a certain condition is met.

CASE Syntax

In general, you construct a CASE comparison query by specifying the constraints for the entire domain over which results are to be produced in the WHERE clause of the main, or outer, query. Then you break the result into subsets with a CASE expression in the select list.

```sql
CASE WHEN search_condition THEN result1 ELSE result2 END AS col_alias
```

where `search_condition` is a logical condition; `result1` is a value to be used when `search_condition` evaluates to “true”; and `result2` is a default value when `search_condition` is “false.”

Note: A CASE expression can take one of two forms: simple or searched. This example uses the searched form. For more details, refer to the SQL Reference Guide. For an example of the simple form, see page 3-19.

About the Example

This query poses the same business question as the previous query in this chapter. In this case, however, the CASE expression is used to produce six different columns in the result set that contain aggregate dollar values—one column for each store.

Usage Notes

In the WHEN condition for the store named Java Judy’s, the apostrophe must be expressed as two single quotes:

```sql
when store_name = 'Java Judy''s'
```

Otherwise, the apostrophe will be interpreted as the closing quote for the character string, and the query will return an “incomplete string” error.
Comparison Queries

Subqueries in the FROM Clause

Question

How did product sales in San Jose during January 1994 compare with annual product sales in the same city during the same year?

Query

```
select product, jan_94_sales, total_94_sales
from
    (select p1.prod_name, sum(dollars)
     from product p1 natural join sales s1
     natural join period d1 natural join store r1
     where d1.year = 1994 and month = 'JAN'
     and r1.city like 'San J%'
     group by p1.prod_name) as sales1(product, jan_94_sales)
natural join
    (select p2.prod_name, sum(dollars) as total_94_sales
     from product p2 natural join sales s2
     natural join period d2 natural join store r2
     where d2.year = 1994 and r2.city like 'San J%'
     group by p2.prod_name) as sales2(product, total_94_sales)
order by product;
```

Result

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>JAN_94_SALES</th>
<th>TOTAL_94_SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>1653.00</td>
<td>21697.50</td>
</tr>
<tr>
<td>Aroma Sheffield Steel Teapot</td>
<td>120.00</td>
<td>1122.00</td>
</tr>
<tr>
<td>Aroma Sounds Cassette</td>
<td>58.50</td>
<td>866.00</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>7.95</td>
<td>2960.15</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>470.85</td>
<td>4470.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>652.00</td>
<td>11375.00</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>352.00</td>
<td>5429.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>608.25</td>
<td>6394.75</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>1936.50</td>
<td>24050.50</td>
</tr>
<tr>
<td>Colombiano</td>
<td>2148.00</td>
<td>22528.50</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>867.50</td>
<td>8590.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>1355.00</td>
<td>17787.50</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>2163.00</td>
<td>35523.50</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>540.50</td>
<td>6608.50</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>393.00</td>
<td>5365.50</td>
</tr>
<tr>
<td>Espresso Machine Italiano</td>
<td>899.55</td>
<td>4397.80</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>2935.50</td>
<td>27362.00</td>
</tr>
<tr>
<td>French Press, 2-Cup</td>
<td>104.65</td>
<td>1196.00</td>
</tr>
<tr>
<td>French Press, 4-Cup</td>
<td>19.95</td>
<td>1109.20</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>440.00</td>
<td>5381.50</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>703.25</td>
<td>7455.50</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A More Flexible Solution: Subqueries in the FROM Clause

A subquery is any query expression enclosed in parentheses that occurs inside another query. A subquery is sometimes referred to as an inner query that operates within an outer query, or as the child query of a parent query.

About the Example

A value is often compared with a sum of a set of values. This example compares product sales in San Jose in January 1994 with product sales in San Jose throughout 1994. This kind of query requires mixed aggregations; therefore, it cannot be written with CASE expressions, which must operate on values within a single group or scope. Instead, subqueries in the FROM clause are used to make the comparison.

Note: Any query that can be expressed as a subquery in the FROM clause can also be expressed as a subquery in the select list, as shown later in this chapter. However, subqueries in the FROM clause generally run faster and are conceptually easier to write.

Usage Notes

This example relies on the flexibility of the query expression in standard SQL to join the results of two subqueries. For detailed information about query expressions, refer to Chapter 7 of the SQL Reference Guide.

Tables derived from the evaluation of subqueries can be joined with other table references. To this end, a subquery in the FROM clause must have a correlation name; however, the list of derived columns is optional. For example, the subqueries in this example evaluate to the following tables:

sales1(product, jan_94_sales)
sales2(product, total_94_sales)

The natural join of these tables (over the Product column) produces an unnamed derived table with three columns—the source of the three select-list items in the main query:

product, jan_94_sales, total_94_sales

For more examples of table joins, refer to Chapter 5, “Joins and Unions.”
Comparison Queries

Calculations and Comparisons

Question

What percentage of annual product sales in San Jose did the January 1994 sales figures in that city represent? What were the top ten products in terms of those percentages?

Query

```sql
select product, jan_94_sales, total_94_sales,
    dec(100 * jan_94_sales/total_94_sales,7,2) as pct_of_94,
    rank(pct_of_94) as rank_pct
from
    (select pl.prod_name, sum(dollars)
     from product p1 natural join sales s1
     natural join period d1 natural join store r1
     where d1.year = 1994 and month = 'JAN'
     and r1.city like 'San J%'
     group by p1.prod_name) as sales1(product, jan_94_sales)
natural join
    (select p2.prod_name, sum(dollars)
     from product p2 natural join sales s2
     natural join period d2 natural join store r2
     where d2.year = 1994
     and r2.city like 'San J%'
     group by p2.prod_name) as sales2(product, total_94_sales)
when rank_pct <= 10
order by product;
```

Result

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>JAN_94_SALES</th>
<th>TOTAL_94_SALES</th>
<th>PCT_OF_94</th>
<th>RANK_PCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Sheffield Steel Teapot</td>
<td>120.00</td>
<td>1122.00</td>
<td>10.69</td>
<td>4</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>470.85</td>
<td>4470.50</td>
<td>10.53</td>
<td>5</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>608.25</td>
<td>6394.75</td>
<td>9.51</td>
<td>9</td>
</tr>
<tr>
<td>Colombiano</td>
<td>2148.00</td>
<td>22528.50</td>
<td>9.53</td>
<td>8</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>867.50</td>
<td>8590.00</td>
<td>10.09</td>
<td>7</td>
</tr>
<tr>
<td>Espresso Machine Italiano</td>
<td>899.55</td>
<td>4397.80</td>
<td>20.45</td>
<td>1</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>2935.50</td>
<td>27362.00</td>
<td>10.72</td>
<td>3</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>701.25</td>
<td>7455.50</td>
<td>9.43</td>
<td>10</td>
</tr>
<tr>
<td>La Antigua</td>
<td>2643.25</td>
<td>22244.50</td>
<td>11.88</td>
<td>2</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>3195.00</td>
<td>31200.00</td>
<td>10.24</td>
<td>6</td>
</tr>
</tbody>
</table>
Calculations with FROM Clause Subqueries

The result set of a comparison query can be used as the source data for various calculations. For example, a product’s monthly total can be expressed as a share of annual sales with a simple percentage calculation:

\[
100 * \frac{monthly\_sales}{annual\_sales}
\]

Simple and complex market, product, and time interval shares or percentages can be calculated with subqueries in the FROM clause.

About the Example

Based on the previous example, this query calculates the monthly sales figures for each product in San Jose as a share or percentage of annual sales for that product in the same city. Using the RANK display function (introduced in Chapter 3), the query also ranks the percentage values and discards all but the top ten products from the result set.

The figures in the Pct_of_94 column do not add up to 100 because these figures represent percentages of one month to a year for individual product sales, not percentages of monthly sales to all annual sales.

Usage Notes

The select list of the main query consists entirely of derived column names, column aliases, and/or expressions that include those names and aliases. For example, the select-list item

\[
\text{dec}(100 * \frac{\text{jan\_94\_sales}}{\text{total\_94\_sales}},7,2) \text{ as pct\_of\_94}
\]

uses columns named in the table that derives from the natural join of the subqueries as the operands of the multiplication (*) and division (/) calculations. In turn, the final select-list item

\[
\text{rank(pct\_of\_94)} \text{ as rank\_pct}
\]

uses the column alias from the previous expression as the argument of the RANK function.

For more examples of RISQL display functions and the use of the WHEN clause, refer to Chapter 3, “Data Analysis.”

Queries that calculate various percentages and performance metrics might require numerous lines of repetitive instructions. Chapter 6 describes how to abbreviate and generalize long SQL statements with RISQL macros.
Comparison Queries

Subqueries in the Select List

Question

During which days of December 1995 were Lotta Latte sales figures at the San Jose Roasting Company lower than the average daily sales figure for the same product at the same store during December 1994?

Display the daily average for 1994 as a separate column.

Query

```
select prod_name, store_name, date, dollars as sales_95,
       (select dec(avg(dollars),7,2)
        from store natural join sales
        natural join product
        natural join period
        where year = 1994
        and month = 'DEC'
        and store_name = 'San Jose Roasting Company'
        and prod_name like 'Lotta%') as avg_94
from store natural join sales
natural join product
natural join period
where prod_name like 'Lotta%'
and store_name = 'San Jose Roasting Company'
and year = 1995
and month = 'DEC'
and dollars <
       (select avg(dollars)
        from store natural join sales
        natural join product
        natural join period
        where year = 1994
        and month = 'DEC'
        and store_name = 'San Jose Roasting Company'
        and prod_name like 'Lotta%');
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>STORE_NAME</th>
<th>DATE</th>
<th>SALES_95</th>
<th>AVG_94</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>San Jose Roasting Company</td>
<td>1995-12-09</td>
<td>153.00</td>
<td>154.72</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>San Jose Roasting Company</td>
<td>1995-12-28</td>
<td>144.50</td>
<td>154.72</td>
</tr>
</tbody>
</table>
Comparisons with Select-List Subqueries

A subquery can occur in the select list of a main query only if it returns one row or no rows. This kind of subquery, a scalar subquery, is useful for spreadsheet-style comparisons in which a series of values returned by the main query is compared to a single value returned by the subquery.

About the Example

This example query returns the daily Lotta Latte sales figures at the San Jose Roasting Company in 1995 in cases where those figures were lower than the average daily sales figure at the same store during 1994. Note that the Avg_94 column contains a single, repeated value representing the 1994 average; this same value would appear in that column regardless of the number of rows in the result set.

The same subquery occurs twice in the main query:
• Once as a column definition in the select list.
• Once as an operand of the less-than operator (<) in a WHERE clause condition.

This query is processed in the following order:
1. The second subquery, which defines the search condition in the WHERE clause of the main query, is executed.
2. The value derived from the second subquery is inserted into the main query’s WHERE clause.
3. The select-list subquery is executed.
4. The main query is executed.

Usage Notes

The DEC scalar function is used on the Avg_94 column of the result set to truncate the average sales figures:

dec(avg(dollars),7,2)
Comparison Queries

Correlated Subqueries

Question

How did individual product sales in San Jose during January 1994 compare with annual sales in the same city during the same year?

Query

```
select pl.prod_name, sum(s1.dollars) as jan_94_sales,
     (select sum(s2.dollars)
      from store r2 natural join sales s2
      natural join product p2 natural join period d2
      where p1.prod_name = p2.prod_name
      and d1.year = d2.year
      and r1.city = r2.city) as total_94_sales
from store r1 natural join sales s1
     natural join product p1
     natural join period d1
where year = 1994 and month = 'JAN'
     and city like 'San J%'
group by p1.prod_name, d1.year, r1.city
order by p1.prod_name;
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>JAN_94_SALES</th>
<th>TOTAL_94_SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>1653.00</td>
<td>21697.50</td>
</tr>
<tr>
<td>Aroma Sheffield Steel Teapot</td>
<td>120.00</td>
<td>1122.00</td>
</tr>
<tr>
<td>Aroma Sounds Cassette</td>
<td>58.50</td>
<td>866.00</td>
</tr>
<tr>
<td>Aroma baseball cap</td>
<td>7.95</td>
<td>2960.15</td>
</tr>
<tr>
<td>Aroma t-shirt</td>
<td>470.85</td>
<td>4470.50</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>652.00</td>
<td>11375.00</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>352.00</td>
<td>5429.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>608.25</td>
<td>6394.75</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>1936.50</td>
<td>24050.50</td>
</tr>
<tr>
<td>Colombiano</td>
<td>2148.00</td>
<td>22528.50</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>867.50</td>
<td>8590.00</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>1355.00</td>
<td>17787.50</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>2163.00</td>
<td>35523.50</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>540.50</td>
<td>6608.50</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>393.00</td>
<td>5365.50</td>
</tr>
<tr>
<td>Espresso Machine Italiano</td>
<td>899.55</td>
<td>4397.80</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>2935.50</td>
<td>27362.00</td>
</tr>
<tr>
<td>French Press, 2-Cup</td>
<td>104.65</td>
<td>1196.00</td>
</tr>
<tr>
<td>French Press, 4-Cup</td>
<td>19.95</td>
<td>1109.20</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>440.00</td>
<td>5381.50</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>703.25</td>
<td>7455.50</td>
</tr>
<tr>
<td>La Antigua</td>
<td>2643.25</td>
<td>22244.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>3195.00</td>
<td>31200.00</td>
</tr>
<tr>
<td>NA Lite</td>
<td>1319.00</td>
<td>27457.00</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Correlated Subqueries in the Select List

Although select-list subqueries must return a single value or no value, they can be executed more than once in reference to results returned by the main query. In this way, such correlated subqueries in the select list can be used to the same effect as subqueries in the FROM clause.

A correlated subquery is closely related to the main query through cross-references to specific values in rows retrieved by the main query. For example, a correlated subquery might reference values in the Month column of the main query; therefore, it will return a new value each time the value of the Month column changes. These cross-references are expressed with table correlation names assigned in the FROM clause.

About the Example

This example presents the same business question as the query on page 4-6, but places the subquery in the select list instead of in the FROM clause. The query compares the sales of products in San Jose during January 1994 with annual sales of products in San Jose in the same year.

To enable the subquery to return a series of values instead of one fixed value, three cross-references correlate the subquery with the main query:

- `p1.prod_name = p2.prod_name`
- `d1.year = d2.year`
- `r1.city = r2.city`

The correlation names `p2`, `d2`, and `r2`, defined in the subquery’s FROM clause, remove ambiguity. Each correlation condition references a specific product, year, and city in the row currently being processed by the main query. These cross-references are sometimes called outer references.

Usage Notes

When there is an aggregate function in the select list of the main query, a GROUP BY clause is required. Column names referenced in a correlation condition of a subquery must appear in the GROUP BY clause of the main query; therefore, the columns

- `d1.year`, `r1.city`

must be listed in the GROUP BY clause, as well as the `Prod_Name` column.

As database identifiers, correlation names must begin with a letter and contain no more than 128 characters. A combination of letters, digits, or underscores can follow the initial letter. (A keyword cannot serve as a database identifier.)
Comparison Queries

Cross-References

Question

What were the monthly sales of Lotta Latte in San Jose during the first three months of 1995 and 1994?

Query

```sql
select q.prod_name, e.month, sum(dollars) as sales_95,
     (select sum(dollars)
      from store t natural join sales s
      natural join product p
      natural join period d
      where d.month = e.month
          and d.year = e.year-1
          and p.prod_name = q.prod_name
          and t.city = u.city)
     as sales_94
from store u natural join product q
     natural join period e natural join sales l
where qtr = 'Q1_95'
    and prod_name like 'Lotta Latte\%'
    and city like 'San J\%'
group by q.prod_name, e.month, e.year, u.city;
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>MONTH</th>
<th>SALES_95</th>
<th>SALES_94</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>JAN</td>
<td>1611.00</td>
<td>3195.00</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>FEB</td>
<td>3162.50</td>
<td>4239.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>MAR</td>
<td>2561.50</td>
<td>2980.50</td>
</tr>
</tbody>
</table>
Cross-References with Expressions

Cross-references in subqueries are not limited to qualified column names; they can also be expressions. For example, the following expressions are valid cross-references:

- `period.year - 1` (previous year)
- `period.quarter - 1` (previous quarter)

These kinds of generalized cross-references simplify the design of applications written for client tools.

About the Example

This query returns the monthly Lotta Latte sales in San Jose during the first three months of both 1995 and 1994. The key to the correlation is that the intended result contains data from the same months but for different years.

The FROM clause of the main query assigns correlation names to all of the joined tables:

```sql
from store u natural join product q
natural join period e natural join sales l
```

The subquery then correlates its execution with the execution of the main query based on the following conditions in the WHERE clause:

```sql
d.month = e.month
d.year = e.year - 1
p.prod_name = q.prod_name
t.city = u.city
```

As the main query retrieves rows, the values of each column in the parent query can change, and the correlation conditions transmit this change to the subquery. The cross-reference to the previous year as $year - 1$ generalizes the subquery by eliminating a constant value (1994).

To change the query to report on other year periods, only the year constraint in the main query need be changed.

Usage Notes

Whenever possible, generalize correlated subqueries and minimize user interaction by using expressions as cross-references. For more information about generalizing queries, refer to Chapter 6, “Macros, Views, and Temporary Tables.”
Comparison Queries

Percentages of Quarter and Year

Question

What were the monthly sales totals in the first quarter of 1994 for products sold in one-pound bags in San Jose? What were the corresponding share of quarter and share of year percentages for each monthly total?

Query

```sql
select pj.prod_name, dj.month, sum(dollars) as mon_sales_94,
    dec(100 * sum(dollars)/
        (select sum(si.dollars)
        from store ri natural join sales si
        natural join product pi
        natural join period di
        where di.qtr = dj.qtr
        and di.year = dj.year
        and pi.prod_name = pj.prod_name
        and pi.pkg_type = pj.pkg_type
        and ri.city = rj.city), 7, 2) as pct_qtr1,
        dec(100 * sum(dollars)/
        (select sum(si.dollars)
        from store ri natural join sales si
        natural join product pi
        natural join period di
        where di.year = dj.year
        and pi.prod_name = pj.prod_name
        and pi.pkg_type = pj.pkg_type
        and ri.city = rj.city), 7, 2) as pct_yr
from store rj natural join sales sj
natural join product pj
natural join period dj
where rj.city = 'San Jose'
    and dj.year = 1994
    and dj.qtr = 'Q1_94'
    and pkg_type = 'One-pound bag'
group by pj.prod_name, dj.month, dj.qtr, dj.year, pj.pkg_type, rj.city
order by pj.prod_name, pct_qtr1 desc;
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>MONTH</th>
<th>MON_SALES_94</th>
<th>PCT_QTR1</th>
<th>PCT_YR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>FEB</td>
<td>688.75</td>
<td>39.91</td>
<td>8.73</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>JAN</td>
<td>594.50</td>
<td>34.45</td>
<td>7.54</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>MAR</td>
<td>442.25</td>
<td>25.63</td>
<td>5.60</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>MAR</td>
<td>742.00</td>
<td>40.61</td>
<td>10.27</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>JAN</td>
<td>600.50</td>
<td>32.86</td>
<td>8.31</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>FEB</td>
<td>484.50</td>
<td>26.51</td>
<td>6.71</td>
</tr>
</tbody>
</table>
...
Calculations with Select-List Subqueries

Monthly percentages for quarters, years, or other time periods can be calculated with a select-list subquery. The main query retrieves the monthly sales figures and two subqueries retrieve the quarterly and yearly sales figures. The monthly percentages require simple calculations: ratios of month-to-quarter sales and month-to-year sales.

About the Example

This example query calculates month-to-quarter and month-to-year sales percentages for selected coffee products sold in San Jose during the first quarter of 1994. After calculating the percentages, the query orders the result table by product and quarterly percentage in descending order.

Usage Notes

Like the previous example, this select-list subquery requires explicit cross-references to correlate the execution of the subquery with the retrieval of new rows by the main query.

In most cases, this kind of comparison query runs faster and is easier to conceptualize as a series of subqueries expressed in the FROM clause. Nonetheless, if the correlated method is your preferred way of expressing the query and the query performs well, there is no need to rewrite it. Both approaches offer the same functionality and produce the same results.
Comparison Queries

Subqueries in the WHERE Clause

Question

During which days in June 1995 were Lotta Latte sales figures at stores in the Chicago district lower than the average daily sales figures for the same product in the same district during June 1994?

Query

```sql
select prod_name, district, date, dollars as sales_95
from market natural join store
    natural join sales
    natural join product
    natural join period
where prod_name like 'Lotta%'
    and district like 'Chic%'
    and year = 1995
    and month = 'JUN'
    and dollars <
    (select avg(dollars)
        from market natural join store
            natural join sales
            natural join product
            natural join period
        where prod_name like 'Lotta%'
            and district like 'Chic%'
            and year = 1994
            and month = 'JUN');
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>DISTRICT</th>
<th>DATE</th>
<th>SALES_95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>Chicago</td>
<td>1995-06-08</td>
<td>76.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>Chicago</td>
<td>1995-06-11</td>
<td>59.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>Chicago</td>
<td>1995-06-17</td>
<td>42.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>Chicago</td>
<td>1995-06-18</td>
<td>76.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>Chicago</td>
<td>1995-06-30</td>
<td>110.50</td>
</tr>
</tbody>
</table>
Comparison Queries

Comparisons with WHERE Clause Subqueries

So far, this chapter has focused on the equivalent functionality but different syntax involved in placing subqueries in the select list or the FROM clause. Subqueries can also be used as search conditions or predicates in the WHERE clause, as a means of pushing complex constraints through to the early stages of the main query’s execution. For example, although you cannot use a set function as part of a simple WHERE clause search condition, you can use a set function in the WHERE clause if it is embedded inside a subquery.

About the Example

This query returns the Lotta Latte sales figures at stores in the Chicago district during 1995 for days on which the sales were lower than the average daily Lotta Latte sales figure for the same city during 1994.

The subquery in this example is scalar—it produces one value. After the subquery has calculated the average dollar figure per day in Chicago in 1994, that single average value is used as a constraint on all the rows returned by the main query. Only those figures for 1995 that were lower than the 1994 average are displayed in the result set; the average figure itself cannot be displayed unless the subquery is moved into the select list or the FROM clause.

Usage Notes

The logical order of query processing dictates that WHERE clause constraints are applied by the server immediately after the tables in the FROM clause have been joined and prior to any calculations with set functions (such as AVG and SUM), RISQL display functions, and so on. Therefore, you cannot use one of those functions in a simple search condition in the WHERE clause.
Comparison Queries

ALL Comparison Predicate

Question

What product registered the highest daily sales total in Hartford, Connecticut, in January 1996?

Query

```
select prod_name, date, dollars
from store natural join sales
    natural join product
    natural join period
where year = 1996
    and city = 'Hartford'
    and month = 'JAN'
    and dollars >= all
        (select dollars
         from store natural join sales
             natural join product
             natural join period
         where year = 1996
             and city = 'Hartford'
             and month = 'JAN');
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>DATE</th>
<th>DOLLARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA Lite</td>
<td>1996-01-24</td>
<td>414.00</td>
</tr>
</tbody>
</table>
Comparison Predicates in Subqueries

The predicates ALL, ANY, SOME, and EXISTS are useful for expressing conditions on groups of values retrieved by a subquery. A comparison predicate states a logical relationship between two values: The comparison is true, false, or unknown with respect to a given row. (The ANY and SOME predicates are synonyms.)

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Evaluates to “true” when</th>
<th>When no value is returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>The comparison is true for all values returned by the subquery.</td>
<td>Evaluates to true.</td>
</tr>
<tr>
<td>SOME, ANY</td>
<td>The comparison is true for at least one of the values returned by the subquery.</td>
<td>Evaluates to false.</td>
</tr>
<tr>
<td>EXISTS</td>
<td>The subquery produces at least one row.</td>
<td>Evaluates to false.</td>
</tr>
</tbody>
</table>

For more information about these predicates, refer to Chapter 3 of the SQL Reference Guide.

About the Example

This query returns the name of the product that recorded the highest daily sales total in Hartford in January 1996 and the specific date when that total was recorded. The query could be rewritten to return the lowest total by replacing the >= operator with a <= operator.

Usage Notes

An alternative (and more concise) way to write this query is to use the RANK function in the WHEN clause:

```sql
select prod_name, date, dollars
from sales natural join period
natural join product
natural join store
where year = 1996
and month = 'JAN'
and city = 'Hartford'
when rank(dollars) = 1;
```

However, RANK queries may yield multiple rows that tie for the rank of 1, while a subquery in the WHERE clause must return one row or no rows.
Comparison Queries

EXISTS Predicate

Question

Which suppliers closed at least one order in March 1996?

Query

```sql
select distinct name as supplier_name
from supplier
where exists
  (select * from orders
   where supplier.supkey = orders.supkey
   and extract(year from close_date) = 1996
   and extract(month from close_date) = 03);
```

Result

```
SUPPLIER_NAME
Aroma East Mfg.
Aroma West Mfg.
Crashing By Design
Espresso Express
Leaves of London
Tea Makers, Inc.
Western Emporium
```
**Comparison Queries**

**EXISTS Predicate**

The EXISTS predicate operates on a subquery and evaluates to “true” or “false.” If it evaluates to “true,” the main query produces a result set. If it evaluates to “false,” the main query returns no rows.

**About the Example**

This query returns the names of each supplier that closed one or more orders with the Aroma Coffee Company in March 1996.

The subquery contains three conditions that test whether any such suppliers exist. The first condition is a join of the Supplier and Orders tables over the Supkey column. The second and third conditions are expressed with the EXTRACT function, which checks for the appropriate dateparts in the Close_Date column of the Orders table. (For a detailed example of this function, refer to “EXTRACT” on page 3-28.)

**Usage Notes**

You could ask the same business question by joining the Supplier, Orders, and Period tables:

```sql
select distinct name as supplier_name
from supplier s, orders o, period p
where s.supkey = o.supkey
  and o.close_date = p.date
  and year = 1996
  and month = 'MAR';
```

Note that this alternative query must join the Orders and Period tables over their Close_Date and Date columns—not their Perkey columns. This is because the Perkey column indicates the date when the orders were entered, which might have been in an earlier month. For example, an order might be entered in the last week of February but received and closed in the first week of March.

This join of the Orders and Period tables is a good example of a join over columns that have no primary-key/foreign-key relationship. The join is possible because the Close_Date and Date columns have comparable datatypes.

The opposite of the EXISTS predicate is NOT EXISTS:

```sql
...
where not exists (select...)
```

For more information about this predicate, refer to the SQL Reference Guide.
Comparison Queries

**SOME or ANY Predicate**

**Question**

Which suppliers have *at some point* provided orders priced at more than $10,000? What were the actual prices of orders closed in March 1996 by those suppliers?

**Query**

```
select name as supplier_name, price
from supplier natural join orders
where extract(year from close_date) = 1996
and extract(month from close_date) = 03
and supplier_name = some
    (select name from supplier
     natural join orders
     where price > 10000)
order by supplier_name;
```

**Result**

<table>
<thead>
<tr>
<th>SUPPLIER_NAME</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma West Mfg.</td>
<td>4425.00</td>
</tr>
<tr>
<td>Espresso Express</td>
<td>30250.00</td>
</tr>
<tr>
<td>Espresso Express</td>
<td>25100.00</td>
</tr>
<tr>
<td>Espresso Express</td>
<td>26400.00</td>
</tr>
<tr>
<td>Espresso Express</td>
<td>22700.00</td>
</tr>
<tr>
<td>Western Emporium</td>
<td>10234.50</td>
</tr>
</tbody>
</table>
Some or ANY Predicate

The SOME and ANY comparison predicates evaluate to “true” when at least one of the values returned by the subquery meets the conditions specified in it. You can use these predicates interchangeably—they are synonyms.

SOME and ANY are useful for retaining rows in the result set when they meet all the conditions specified in the inner query but not all the conditions in the outer query. For example, the outer query might request a list of suppliers that shipped orders in a specific month, regardless of their price, while the inner query might request a list of suppliers that have shipped orders that cost more than a specific amount in any month.

About the Example

This example uses the SOME predicate to return a list of suppliers and order prices:

- The subquery returns a list of suppliers that have supplied at least one order that cost more than $10,000.
- The main query takes that list of suppliers and matches it with records of orders closed in March 1996, with no constraint on the price of each order.

The last row in the result set shows that there was an order supplied by Aroma West Mfg. in March 1996 that cost $4,425.00. The presence of this row indicates that at some other point in time, Aroma West Mfg. supplied at least one order that cost more than $10,000.

Usage Notes

The EXTRACT function is used in the same way in this query as in the example of the EXISTS predicate on page 4-22.

For more information about these predicates, refer to the SQL Reference Guide.
Comparison Queries

Summary

This chapter described how to write queries that compare data and display the results in a readable format. Various approaches were illustrated:

- CASE expressions
- FROM clause subqueries
- Select-list subqueries, including correlated subqueries
- WHERE clause subqueries

The chapter ended with examples of the ALL, ANY, SOME, and EXISTS comparison predicates, which can be used as conditions on subquery results.

Some of the more complex examples showed how to include calculations in comparison queries, such as percentages that represent share of quarter or share of year.

Note: In general, query performance is faster when comparison queries use CASE expressions rather than subqueries. If subqueries are necessary, however, the preferred method is to use the FROM clause rather than the select list.
This chapter describes two different ways to combine data from different tables:

- By joining the tables
- By using the UNION, EXCEPT, and INTERSECT operators

The first part of this chapter presents several examples of inner and outer joins.

The second part illustrates how to combine data from different tables by using UNION, EXCEPT, and INTERSECT operators, which take the intermediate result set from one query expression and combine it with the result set from another query expression.
Joins and Unions

Join of Two Tables

<table>
<thead>
<tr>
<th>State Table</th>
<th>Region Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>CITY</td>
<td>STATE</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
</tr>
</tbody>
</table>

Query

select * from state, region;

Cartesian Product
(join predicate not specified)

<table>
<thead>
<tr>
<th>CITY</th>
<th>STATE</th>
<th>CITY</th>
<th>AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>

Query

select * from state, region
where state.city = region.city;

Subset of Cartesian Product
(join predicate specified)

<table>
<thead>
<tr>
<th>STATE:CITY</th>
<th>STATE:STATE</th>
<th>REGION:CITY</th>
<th>REGION:AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
</tbody>
</table>

5-2 SQL Self-Study Guide
Inner Joins

Most queries join information from different tables. Any two tables can be
joined over columns with comparable datatypes; joins are not dependent on
primary key–foreign key relationships.

Cartesian Product

When two or more tables are referenced in the FROM clause of a query, the
warehouse server joins the tables. If neither the FROM clause nor the WHERE
clause specifies a predicate for the join, the server computes a Cartesian
product that contains \( m \times n \) rows, where \( m \) is the number of rows in the first
table and \( n \) is the number of rows in the second table. This product is the set of
all possible combinations formed by concatenating a row from the first table
with a row from the second table.

Note: If the OPTION CROSS JOIN parameter in the rbw.config file is set to OFF
(the default), cross-join queries are not executed.

Subset of the Cartesian Product

If tables are explicitly joined over columns with comparable datatypes, the
server computes a subset of the Cartesian product. This subset contains only
those rows where the values in the joining columns match. For the duration of
the query, the subset functions as a derived table that can be joined with other
tables or the results of other query expressions.

About the Example

The State and Region tables both contain City columns, which are specified as
the joining columns in the WHERE clause. Consequently, only those rows of the
Cartesian product that have matching City keys are displayed in the result. In
this example, the result table contains only 2 rows whereas the full Cartesian
product of these two tables contains 9 rows.

The joining columns could alternatively be specified in the FROM clause, as
discussed on page 5-5.

Note: The tables used on the facing page are not part of the Aroma database;
Aroma tables are used in the examples later in this chapter.
**Joins and Unions**

**Different Ways to Join Tables**

**Question**

How long did the Christmas special promotion run in 1994 and 1995? What were the total sales for products sold on that promotion in each year, and what was the average sales total per day in each year?

**Query 1**

```sql
select promo_desc, year, sum(dollars) as sales,
       datediff(day, end_date, start_date)+1 as days_on_promo,
       string(sales/days_on_promo, 7, 2) as per_day
from period natural join sales
     natural join promotion
where promo_desc like 'Christmas%'
and year in (1994, 1995)
group by promo_desc, year, days_on_promo;
```

**Query 2**

```sql
select promo_desc, year, sum(dollars) as sales,
       datediff(day, end_date, start_date)+1 as days_on_promo,
       string(sales/days_on_promo, 7, 2) as per_day
from period join sales on period.perkey = sales.perkey
     join promotion on promotion.promokey = sales.promokey
where promo_desc like 'Christmas%'
and year in (1994, 1995)
group by promo_desc, year, days_on_promo;
```

**Query 3**

```sql
select promo_desc, year, sum(dollars) as sales,
       datediff(day, end_date, start_date)+1 as days_on_promo,
       string(sales/days_on_promo, 7, 2) as per_day
from period join sales using(perkey)
     join promotion using(promokey)
where promo_desc like 'Christmas%'
and year in (1994, 1995)
group by promo_desc, year, days_on_promo;
```

**Three Queries—Same Result**

<table>
<thead>
<tr>
<th>PROMO_DESC</th>
<th>YEAR</th>
<th>SALES</th>
<th>DAYS_ON_PROMO</th>
<th>PER_DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christmas special</td>
<td>1994</td>
<td>690.00</td>
<td>31</td>
<td>22.25</td>
</tr>
<tr>
<td>Christmas special</td>
<td>1995</td>
<td>1230.00</td>
<td>31</td>
<td>39.67</td>
</tr>
</tbody>
</table>
Joins in the FROM Clause

There are three ways to explicitly join tables in the FROM clause:

- Natural join
- Join over named columns (USING syntax)
- Join over predicate (ON syntax)

About the Example

This query joins the Promotion, Period, and Sales tables over columns with identical names; therefore, it can be abbreviated with the NATURAL JOIN syntax, as shown in Query 1. Queries 2 and 3 show alternative methods of specifying inner equijoins in the FROM clause. The result set is the same in all three cases; however, the ON and USING join specifications retain both joining columns in their intermediate result sets, whereas the NATURAL JOIN specification combines each pair of joining columns into one column.

Note the use of scalar functions in this query:

- The DATEDIFF function is used to calculate the duration of the Christmas promotion. This function is discussed in detail on page 3-27.
- The STRING function is used to scale the Per_Day column values down to a precision of two decimal places; without this function, the expression `sales/promo_days` would return long-numeric values.

Usage Notes

Natural joins operate on all pairs of columns that have identical names and should be used with caution; otherwise, tables might be inadvertently joined over columns that happen to have the same name but were not intended to participate in the join.

In the retail schema of the Aroma database, all the primary key–foreign key relationships are based on columns with the same name, so natural joins are effective for most queries that involve the Sales table and its dimensions.

For an example of a join over non-primary key/foreign key columns, refer to “Calculating Elapsed Days: DATEDIFF” on page 3-27. For a complete discussion of join syntax, refer to Chapter 7 of the SQL Reference Guide.
**System Table Join**

**Question**

What are the names of the segments and physical storage units (PSUs) used to store the Aroma Sales table?

**Query**

```sql
select segname as storage, location as psu_location,
       tname as table_name
from rbw_storage join rbw_segments on
                 rbw_storage.segname = rbw_segments.name
where table_name = 'SALES'
order by psu_location;
```

**Result**

<table>
<thead>
<tr>
<th>STORAGE</th>
<th>PSU_LOCATION</th>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_SEGMENT_23</td>
<td>dfltseg23_psu1</td>
<td>SALES</td>
</tr>
<tr>
<td>DAILY_DATA1</td>
<td>sales_psu1</td>
<td>SALES</td>
</tr>
<tr>
<td>DAILY_DATA1</td>
<td>sales_psu2</td>
<td>SALES</td>
</tr>
<tr>
<td>DAILY_DATA2</td>
<td>sales_psu3</td>
<td>SALES</td>
</tr>
<tr>
<td>DAILY_DATA2</td>
<td>sales_psu4</td>
<td>SALES</td>
</tr>
</tbody>
</table>
Joining System Tables

Warehouse administrators need to know the relationships between different database objects, such as tables and indexes or tables and segments. To facilitate access to this kind of information, Red Brick Warehouse system tables can be joined in the same way as all other database tables.

About the Example

This query joins two system tables to identify the names of both default and user-defined segments and their associated PSUs for the Sales table in the Aroma database.

Usage Notes

The WHERE clause condition

```
table_name = 'SALES'
```

must use uppercase for SALES; otherwise, no matching rows are found.

For detailed information about system tables, table segmentation, and so on, refer to the Warehouse Administrator’s Guide.
Self-Joins

Question

Which products in the Product table have the same names but different types of packaging?

Query

```sql
select a.prod_name as products,
       a.pkg_type
from product a, product b
where a.prod_name = b.prod_name
     and a.pkg_type <> b.pkg_type
order by products, a.pkg_type;
```

Result

<table>
<thead>
<tr>
<th>PRODUCTS</th>
<th>PKG_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>No pkg</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>No pkg</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>No pkg</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>No pkg</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>Colombiano</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>No pkg</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>No pkg</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>No pkg</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>No pkg</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>No pkg</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>One-pound bag</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>No pkg</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>No pkg</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Qtr-pound bag</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Joining a Table to Itself

There is no requirement that the tables being joined in a query be distinct; you can join any table to itself as long as you give each table reference a different name. Self-joins are useful for discovering relationships between different columns of data in the same table.

About the Example

This query joins the Product table to itself over the Prod_Name column, using the aliases \( a \) and \( b \) to distinguish the table references:

```sql
from product a, product b
```

The self-join compares Product table \( a \) to Product table \( b \) to find rows where the product names match but the package types differ:

```sql
where a.prod_name = b.prod_name
    and a.pkg_type <> b.pkg_type
```

The result set consists of a list of each pair of identically named products and their individual package types.
**Joins and Unions**

**Outer Join of Two Tables**

*Query (left outer join)*

```sql
select * from state left outer join region
  on state.city = region.city;
```

**Result**

<table>
<thead>
<tr>
<th>STATE:CITY</th>
<th>STATE:STATE</th>
<th>REGION:CITY</th>
<th>REGION:AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

*Query (right outer join)*

```sql
select * from state right outer join region
  on state.city = region.city;
```

**Result**

<table>
<thead>
<tr>
<th>STATE:CITY</th>
<th>STATE:STATE</th>
<th>REGION:CITY</th>
<th>REGION:AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>

*Query (full outer join)*

```sql
select * from state full outer join region
  on state.city = region.city;
```

**Result**

<table>
<thead>
<tr>
<th>STATE:CITY</th>
<th>STATE:STATE</th>
<th>REGION:CITY</th>
<th>REGION:AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacksonville</td>
<td>FL</td>
<td>Jacksonville</td>
<td>South</td>
</tr>
<tr>
<td>Miami</td>
<td>FL</td>
<td>Miami</td>
<td>South</td>
</tr>
<tr>
<td>Nashville</td>
<td>TN</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>New Orleans</td>
<td>South</td>
</tr>
</tbody>
</table>

**Note:** These examples use the tables introduced on page 5-2.
Joins and Unions

Outer Joins

In most cases, tables are joined according to search conditions that find only the rows with matching values; this type of join is known as an inner equijoin. In some cases, however, decision-support analysis requires outer joins, which retrieve both matching and non-matching rows, or non-equijoins, which express, for example, a greater-than or less-than relationship.

An outer join operation returns all the rows returned by an inner join plus all the rows from one table that do not match any row from the other table. An outer join can be “left,” “right,” or “full,” depending on whether rows from the left, right, or both tables are retained. The first table listed in the FROM clause is referred to as the left table and the second as the right table. For all three types of outer join, NULLs are used to represent empty columns in rows that do not match.

Syntax

An outer join between two tables can be specified in the FROM clause with the OUTER JOIN keywords followed by the ON subclause:

```
FROM table_1 LEFT|RIGHT|FULL OUTER JOIN table_2
ON table_1.column = table_2.column
```

as shown in the examples on the facing page.

There are other ways to specify outer join predicates in the FROM clause; for details, refer to Chapter 7 of the SQL Reference Guide.

About the Examples

• The result of the left outer join contains every row from the State table and all matching rows in the Region table. Rows found only in the Region table are not displayed.
• The result of the right outer join contains every row from the Region table and all matching rows from the State table. Rows found only in the State table are not displayed.
• The result of the full outer join contains those rows that are unique to each table, as well as those that are common to both.
**Joins and Unions**

**Fact-to-Fact Join**

**Question**
What were the prices paid per line item and/or per full order for order numbers 3619 through 3626?

**Query**
```sql
select coalesce(o.order_no, l.order_no) as order_num, 
    order_type, o.price as full_cost, 
    l.price as line_cost
from orders o left outer join line_items l
    on o.order_no = l.order_no
    join period on o.perkey = period.perkey
where o.order_no between 3619 and 3626
order by order_num;
```

**Result**

<table>
<thead>
<tr>
<th>ORDER_NUM</th>
<th>ORDER_TYPE</th>
<th>FULL_COST</th>
<th>LINE_COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>725.25</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3619</td>
<td>Tea</td>
<td>4325.25</td>
<td>400.00</td>
</tr>
<tr>
<td>3620</td>
<td>Tea</td>
<td>4325.25</td>
<td>NULL</td>
</tr>
<tr>
<td>3621</td>
<td>Spice</td>
<td>10234.50</td>
<td>10234.50</td>
</tr>
<tr>
<td>3622</td>
<td>Spice</td>
<td>10234.50</td>
<td>10234.50</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>400.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>400.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>500.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>450.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>500.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>275.00</td>
</tr>
<tr>
<td>3623</td>
<td>Hardware</td>
<td>4425.00</td>
<td>650.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>1250.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>400.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>500.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>450.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>400.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>500.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>275.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>650.00</td>
</tr>
<tr>
<td>3624</td>
<td>Hardware</td>
<td>4425.00</td>
<td>1250.00</td>
</tr>
<tr>
<td>3625</td>
<td>Clothing</td>
<td>3995.95</td>
<td>2500.00</td>
</tr>
<tr>
<td>3625</td>
<td>Clothing</td>
<td>3995.95</td>
<td>1495.95</td>
</tr>
<tr>
<td>3626</td>
<td>Hardware</td>
<td>16500.00</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Left Outer Join

Outer joins are often used to join fact tables, as a means of comparing related sets of measurements that cannot be queried from a single table.

About the Example

The Orders and Line_Items tables store related facts; however, the line-item detail information for an order might be loaded into the warehouse sometime after the order information is loaded. If an analyst wants to see both order and line-item prices, if available, or just the order prices when no line-item price is available, an outer join is required.

The query returns the prices of both full orders and line items; if the line item prices are unavailable, the full order price is still displayed and the Line_Price and Line_Orders columns contain NULLs. This effect is achieved by using a left outer join, the Orders table being treated as the left table.

The COALESCE function is used to derive one column heading in the report from two columns in the select list:

```sql
coalesce(o.order_no, l.order_no) as order_num
```

Because either column might be NULL, the COALESCE function will produce the value of the non-NULL column. Without this function, the result set would consist of duplicate columns for the order numbers.

Usage Notes

There are different ways to express outer join conditions with standard SQL. For details, refer to Chapter 7 of the SQL Reference Guide.

This example uses the tables in the Aroma purchasing schema, which is described in detail in Appendix A, “The Complete Aroma Database.”
**Joins and Unions**

**Fact-to-Fact Join**

**Question**

In weeks 12 and 13 of 1996, how did revenues from sales compare with expenditures on orders?

**Query**

```sql
select date, extract(week from date) as wk_no, prices, sales
from
  ((select d1.date, sum(price)
  from orders natural join period d1
  where d1.year = 1996 and d1.week in (12, 13)
  group by d1.date) as t1
full outer join
  (select d2.date, sum(dollars)
  from sales natural join period d2
  where d2.year = 1996 and d2.week in (12, 13)
  group by d2.date) as t2
  on t1.date = t2.date)
(order_date, prices, date, sales)
order by wk_no, date
break by wk_no summing prices, sales;
```

**Result**

<table>
<thead>
<tr>
<th>Date</th>
<th>WK_NO</th>
<th>Prices</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996-03-17</td>
<td>12</td>
<td>NULL</td>
<td>9177.90</td>
</tr>
<tr>
<td>1996-03-18</td>
<td>12</td>
<td>NULL</td>
<td>7412.65</td>
</tr>
<tr>
<td>1996-03-19</td>
<td>12</td>
<td>NULL</td>
<td>8620.25</td>
</tr>
<tr>
<td>1996-03-20</td>
<td>12</td>
<td>27025.25</td>
<td>8417.95</td>
</tr>
<tr>
<td>1996-03-21</td>
<td>12</td>
<td>NULL</td>
<td>8230.05</td>
</tr>
<tr>
<td>1996-03-22</td>
<td>12</td>
<td>NULL</td>
<td>9870.20</td>
</tr>
<tr>
<td>1996-03-23</td>
<td>12</td>
<td>NULL</td>
<td>8757.50</td>
</tr>
<tr>
<td>NULL</td>
<td>12</td>
<td>27025.25</td>
<td>60486.50</td>
</tr>
<tr>
<td>1996-03-24</td>
<td>13</td>
<td>NULL</td>
<td>8394.25</td>
</tr>
<tr>
<td>1996-03-25</td>
<td>13</td>
<td>3995.95</td>
<td>10046.90</td>
</tr>
<tr>
<td>1996-03-26</td>
<td>13</td>
<td>NULL</td>
<td>10404.15</td>
</tr>
<tr>
<td>1996-03-27</td>
<td>13</td>
<td>36275.25</td>
<td>9237.00</td>
</tr>
<tr>
<td>1996-03-28</td>
<td>13</td>
<td>NULL</td>
<td>8509.55</td>
</tr>
<tr>
<td>1996-03-29</td>
<td>13</td>
<td>16500.00</td>
<td>7956.10</td>
</tr>
<tr>
<td>1996-03-30</td>
<td>13</td>
<td>NULL</td>
<td>9462.20</td>
</tr>
<tr>
<td>NULL</td>
<td>13</td>
<td>56771.20</td>
<td>64010.15</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>83796.45</td>
<td>124496.65</td>
</tr>
</tbody>
</table>
**Joins and Unions**

**Full Outer Join with ORDER BY, BREAK BY**

Full outer joins return results that include rows from the left and right tables, whether or not they contain matching values in the joining columns. In the result set, the columns for which no match was found contain NULLs.

**About the Example**

The Sales and Line_Items tables store different sets of facts but share two dimension tables—Product and Period. To create a report of orders and sales over a given period of time, you can inner-join each fact table to the Period table, then outer-join the results of the inner joins. One way to do this is to use subqueries in the FROM clause.

The first subquery evaluates to a table named \( t1 \), the second to a table named \( t2 \). Table \( t3 \) is the result of the full outer join of \( t1 \) and \( t2 \). Table \( t3 \) consists of four named columns:

\[
t3(\text{order
date}, \text{prices}, \text{date}, \text{sales})
\]

The select list of the main query references three of these columns—Prices, Date, and Sales. A fourth column in the select list—Wk_No—is extracted from the Date column with the EXTRACT scalar function:

\[
\text{extract(week from date) as wk_no}
\]

The ORDER BY clause and its BREAK BY subclause sort the data by week and date, then display subtotals for each week for both the Prices column and the Sales column. The last row of the result set displays grand totals.

**Usage Notes**

Table aliases are required in this query because the same table name cannot be repeated in the FROM clause. For example, the Period table is referenced as \( d1 \) in one join specification and as \( d2 \) in another.

Any column referenced in a BREAK BY clause must also be listed in the ORDER BY clause. For more information about these clauses, refer to Chapter 7 of the *SQL Reference Guide*.

The example on the facing page is similar in construction to some of the FROM clause subqueries in Chapter 4, “Comparison Queries.”

For more examples of queries that use datetime scalar functions, refer to Chapter 3, “Data Analysis.”
Joins and Unions

OR Versus UNION

Question

What were the total sales in week 52 of 1995 for all Aroma stores classified as “Medium”? What were the totals during the same period for “Large” stores?

Query with OR Condition

```sql
select store_name as store, store_type as size, state,
       sum(dollars) as sales
from period t join sales s on t.perkey = s.perkey
    join store r on r.storekey = s.storekey
where (store_type = 'Medium' or store_type = 'Large')
    and year = 1995
    and week = 52
group by store, size, state
order by size, store;
```

UNION Query

```sql
select store_name as store, store_type as size, state,
       sum(dollars) as sales
from period t join sales s on t.perkey = s.perkey
    join store r on r.storekey = s.storekey
where store_type = 'Medium'
    and year = 1995
    and week = 52
group by store, size, state
union
select store_name as store, store_type as size, state,
       sum(dollars) as sales
from period t join sales s on t.perkey = s.perkey
    join store r on r.storekey = s.storekey
where store_type = 'Large'
    and year = 1995
    and week = 52
group by store, size, state
order by size, store;
```

Two Queries—Same Result

<table>
<thead>
<tr>
<th>STORE</th>
<th>SIZE</th>
<th>STATE</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beaches Brew</td>
<td>Large</td>
<td>CA</td>
<td>2794.30</td>
</tr>
<tr>
<td>Miami Espresso</td>
<td>Large</td>
<td>FL</td>
<td>5302.25</td>
</tr>
<tr>
<td>Olympic Coffee Company</td>
<td>Large</td>
<td>GA</td>
<td>2938.35</td>
</tr>
<tr>
<td>San Jose Roasting Company</td>
<td>Large</td>
<td>CA</td>
<td>3456.65</td>
</tr>
<tr>
<td>Beans of Boston</td>
<td>Medium</td>
<td>MA</td>
<td>3208.00</td>
</tr>
<tr>
<td>Cupertino Coffee Supply</td>
<td>Medium</td>
<td>CA</td>
<td>2771.25</td>
</tr>
<tr>
<td>Java Judy’s</td>
<td>Medium</td>
<td>AZ</td>
<td>3174.25</td>
</tr>
<tr>
<td>Moulin Rouge Roasting</td>
<td>Medium</td>
<td>LA</td>
<td>2670.25</td>
</tr>
<tr>
<td>Texas Teahouse</td>
<td>Medium</td>
<td>TX</td>
<td>4247.25</td>
</tr>
</tbody>
</table>
Combining Result Sets: **UNION**

You can use the UNION, EXCEPT, and INTERSECT operators to combine the output of two or more query expressions into a single set of rows and columns. The server evaluates each query expression independently, then combines the output, displaying column headings from the *first* expression. The server eliminates duplicate result rows unless you specify the ALL keyword.

**UNION, INTERSECT, EXCEPT**

```
query_expression UNION | INTERSECT | EXCEPT [ALL] query_expression
[ORDER BY order_list]
[SUPPRESS BY suppress_list];
```

where *query_expression* is any join or non-join query expression, as defined in Chapter 7 of the SQL Reference Guide. If SUPPRESS BY and ORDER BY clauses are used, they must reference columns from the select list of the first query expression.

**About the Example**

The same business question can be answered by either specifying an OR condition in a single SELECT statement or combining two query expressions with a UNION operator.

Using the OR connective is easier in this simple example, but in some cases a UNION operation improves query performance. For example, suppose your query requires that you access data in two large fact tables: The outer join operation required by a single query might require more processing than using a UNION operation to combine the results of two query expressions.

The ORDER BY clause must reference the column aliases, not the column names, defined in the select list of the first query expression:

```
order by size, store
```

**Usage Notes**

UNION, INTERSECT, and EXCEPT queries must be symmetrical—that is, the select lists on both sides of the UNION operator must match in terms of both the number of columns and their order. Corresponding columns must have the same, or comparable, datatypes, although they may have different names.

Multiple UNION, INTERSECT, and EXCEPT operators can be used in a single statement; operations are evaluated from left to right unless you specify precedence with parentheses.
Joins and Unions

**INTERSECT Operation**

**Question**

Which bulk tea products sold on promotion in San Jose in 1996 were also sold on promotion in New Orleans in 1995? What promotions were run on those products?

**Query**

```sql
select prod_name as tea_name, promo_desc
from sales natural join class
  natural join product
  natural join store
  natural join period
  natural join promotion
where city = 'San Jose'
  and year = 1996
  and class_desc like 'Bulk tea%'
intersect
select prod_name, promo_desc
from sales natural join class
  natural join product
  natural join store
  natural join period
  natural join promotion
where city = 'New Orleans'
  and year = 1995
  and class_desc like 'Bulk tea%'
  and promo_desc not like 'No promo%'
order by promo_desc;
```

**Result**

<table>
<thead>
<tr>
<th>TEA_NAME</th>
<th>PROMO_DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irish Breakfast</td>
<td>Aroma catalog coupon</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Aroma catalog coupon</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Store display</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Temporary price reduction</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Temporary price reduction</td>
</tr>
</tbody>
</table>
**INTERSECT—Finding Common Rows**

You can use the INTERSECT operator to return only those rows that are common to the results returned by two or more query expressions.

**About the Example**

This example finds the intersection of two query expressions—one that returns a list of bulk tea products sold on promotion in San Jose in 1996 and one that returns a similar list for New Orleans in 1995. The INTERSECT operator eliminates all rows that are not found in both preliminary result sets.

**Usage Notes**

The results of UNION, EXCEPT, and INTERSECT operations derive column headings only from the first query expression in the query; therefore, the column alias Tea_Name need only be specified in the first query expression.
**INTERSECT Operation Inside Subquery**

**Question**

Of the products that were ordered in March 1996, which ones were also sold at the Coffee Connection store during that month?

What did orders of these products cost in that month?

What was the total revenue (sum of sales dollars) for those products in the entire Northern region during that month?

**Query**

```sql
select product, cost_of_orders, revenue_north
from (select prod_name
    from product natural join sales natural join period
    natural join store
    where year = 1996 and month = 'MAR'
    and store_name = 'Coffee Connection'
    intersect
    select prod_name
    from product natural join line_items natural join period
    where year = 1996 and month = 'MAR') as p(product)
natural join
(select prod_name, sum(price)
    from product natural join line_items natural join period
    where year = 1996 and month = 'MAR'
    group by prod_name) as c(product, cost_of_orders)
natural join
(select prod_name, sum(dollars)
    from product natural join sales natural join period
    natural join store natural join market
    where year = 1996 and month = 'MAR' and region = 'North'
    group by prod_name) as r(product, revenue_north)
order by product;
```

**Result**

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>COST_OF_ORDERS</th>
<th>REVENUE_NORTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroma Roma</td>
<td>7300.00</td>
<td>3190.00</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>7300.00</td>
<td>3975.50</td>
</tr>
<tr>
<td>Colombiano</td>
<td>7300.00</td>
<td>3907.50</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>8500.00</td>
<td>6081.25</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>7300.00</td>
<td>4218.50</td>
</tr>
<tr>
<td>La Antigua</td>
<td>7300.00</td>
<td>3510.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>7300.00</td>
<td>4273.00</td>
</tr>
<tr>
<td>NA Lite</td>
<td>7300.00</td>
<td>6480.00</td>
</tr>
<tr>
<td>Veracruzano</td>
<td>7300.00</td>
<td>4055.00</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>7300.00</td>
<td>6896.50</td>
</tr>
</tbody>
</table>
INTERSECT of Fact Table Data

The UNION, INTERSECT, and EXCEPT operators are useful for querying tables that contain similar or comparable sets of facts.

About the Example

This query contains three subqueries in the FROM clause. The function of the INTERSECT operator inside the first subquery is to produce a list of products that were ordered in February 1996 as well as sold at the Coffee Connection store in the same month. This is done by placing the INTERSECT operator between query expressions that join two different fact tables—Sales and Line_Items.

The second subquery produces the sum of the order prices for March 1996 for the list of products produced by the first subquery.

The third subquery produces the sum of the sales dollars for the same list of products during the same month, but for the whole Northern region.

Usage Notes

The query on the facing page is similar in construction to the examples of FROM clause subqueries in Chapter 4, “Comparison Queries.” The select list of the main query consists entirely of columns named in the tables derived from the subqueries.
Joins and Unions

**EXCEPT Operation**

**Question**

What were the total 1995 revenues for stores in California cities that are not defined as HQ cities in the Market table?

**Query**

```sql
select city, store_name, sum(dollars) as sales_95
from (select city
    from store
    where state = 'CA'
    except
    select hq_city
    from market
    where hq_state = 'CA')
    as except_cities(city)
natural join store
natural join sales
natural join period
where year = 1995
group by city, store_name
order by sales_95 desc;
```

**Result**

<table>
<thead>
<tr>
<th>CITY</th>
<th>STORE_NAME</th>
<th>SALES_95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cupertino</td>
<td>Cupertino Coffee Supply</td>
<td>196439.75</td>
</tr>
<tr>
<td>Los Gatos</td>
<td>Roasters, Los Gatos</td>
<td>175048.75</td>
</tr>
</tbody>
</table>
**EXCEPT—Finding the Exceptions in Two Result Sets**

The EXCEPT operator finds the exceptions in (or the difference between) the results of two query expressions. For example, an EXCEPT operation could compare lists of products sold at two stores, eliminate all the products sold at both, and retain only those products sold exclusively at the store specified in the first query expression.

**About the Example**

In this example, the function of the EXCEPT operator is to select those California cities that are defined in the City column of the Store table but not in the Hq_City column of the Market table.

This query uses a subquery in the FROM clause to produce a derived table of cities that can be joined with the Sales, Store, and Period tables. The table derived from the subquery is given a correlation name and one column name:

```sql
except_cities(city)
```

This derived table can be joined with the Store table using a natural join over the City column.

**Usage Notes**

To test the outcome of the EXCEPT operation, you could run the subquery in this example as a query in its own right:

```sql
select city
from store
where state = 'CA'
except
select hq_city
from market
where hq_state = 'CA';
```

CITY  
Cupertino  
Los Gatos

For more examples of subqueries, refer to Chapter 4.
**Joins and Unions**

**Summary**

This chapter described:

- How to join tables.
- How to combine the results of two independent query expressions by using the UNION, INTERSECT, and EXCEPT operators.

**Joining Tables**

When a query’s FROM clause lists two or more tables, the server joins the tables. The server can perform both inner and outer joins between any two tables on any two columns with comparable datatypes. There are various ways to write join specifications, using either the FROM clause or the WHERE clause.

**UNION, INTERSECT, and EXCEPT Operators**

```
query_expression
UNION | INTERSECT | EXCEPT [ALL]
query_expression
[ORDER BY order_list]
[SUPPRESS BY suppress_list];
```
This chapter shows how to simplify SQL statements with RISQL macros. A macro is an abbreviation for a complex expression. Macros allow you to write concise, reusable SQL statements.

This chapter also presents simple examples of two other means of simplifying data retrieval—views and temporary tables.

The examples in this chapter show how to:

• Abbreviate a lengthy or frequently used expression or query by writing a macro.
• Write a macro that contains other macros.
• Write generalized macros that use parameters.
• Create and query a view.
• Create, populate, and query a temporary table.
Basic Macros

Question

What were the total sales of tea products during 1995?

CREATE MACRO Statement

create macro tea_products as
  (pt.classkey = 2 or pt.classkey = 5);

Query

select prod_name,
  case pt.classkey when 2 then 'Bulk Tea'
    when 5 then 'Pkg Tea' end as class,
  sum(dollars) as sales_95
from product pt join sales sa
  on pt.classkey = sa.classkey
  and pt.prodkey = sa.prodkey
join period pd on pd.perkey = sa.perkey
where tea_products
  and year = 1995
group by prod_name, pt.classkey
order by sales_95 desc;

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>CLASS</th>
<th>SALES_95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Darjeeling Special</td>
<td>Bulk Tea</td>
<td>80610.50</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Pkg Tea</td>
<td>51266.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Bulk Tea</td>
<td>42329.00</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Bulk Tea</td>
<td>34592.75</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Bulk Tea</td>
<td>27763.75</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Pkg Tea</td>
<td>27192.50</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Bulk Tea</td>
<td>25848.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Bulk Tea</td>
<td>24594.00</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Pkg Tea</td>
<td>24232.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Bulk Tea</td>
<td>23269.50</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Bulk Tea</td>
<td>22326.00</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Bulk Tea</td>
<td>21964.00</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Bulk Tea</td>
<td>21584.50</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Pkg Tea</td>
<td>20084.00</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Pkg Tea</td>
<td>18955.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Pkg Tea</td>
<td>17031.50</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Pkg Tea</td>
<td>16783.25</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Pkg Tea</td>
<td>16773.25</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Pkg Tea</td>
<td>16724.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Pkg Tea</td>
<td>16108.00</td>
</tr>
</tbody>
</table>
Basic Macros

A macro is an abbreviation for a complex expression. For example, you can define a short, meaningful name for a numeric code and reference the code by its macro name rather than by a string of digits. In the same way, you can define a macro for a complete set of conditions and reference those conditions in a query with the macro name. The set of conditions might be a complete SELECT statement or a specific clause in a SELECT statement, for example.

A macro name is a character string that begins with a letter and does not exceed 128 characters. The warehouse server is not sensitive to case: *share* and *SHARE* are equivalent. RISQL keywords cannot be used as macro names.

**CREATE MACRO Syntax**

```sql
CREATE MACRO macro_name AS definition;
```

where:

- `macro_name` A unique name that you specify in an SQL statement to call the macro definition.
- `definition` A complete or partial SQL statement. Only one complete SQL statement can occur in the definition.

Existing macros must be dropped before a macro of the same name can be defined.

```sql
DROP MACRO macro_name;
```

The CREATE MACRO and DROP MACRO commands have additional optional parameters. For details, refer to the *SQL Reference Guide*.

**About the Example**

The macro `tea_products` is based on the knowledge that Classkey values 2 and 5 always refer to bulk tea and packaged tea products, respectively. The Classkey values are queried from the Product table rather than the Class table to simplify the joins in the query; a CASE expression is used to convert the Classkey values to meaningful text values.

The query calculates the 1995 sales totals for all tea products using the macro name. When the warehouse server interprets the query, the macro name is replaced with the character string defined in the CREATE MACRO statement.

The parentheses around the macro definition are required in this example; they force the correct evaluation of the logical operators defined in the macro.
Embedded Macros

Question

What were the total sales of tea products during 1995?

CREATE MACRO Statements

create macro case_tea as
    case pt.classkey when 2 then 'Bulk Tea'
    when 5 then 'Pkg Tea'
    end as class;

create macro tea_totals as
    select prod_name, case_tea, sum(dollars) as sales_95
    from product pt join sales sa
    on pt.classkey = sa.classkey and pt.prodkey = sa.prodkey
    join period pd on pd.perkey = sa.perkey
    where tea_products
    and year = 1995
    group by prod_name, class
    order by sales_95 desc;

Query

tea_totals;

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>CLASS</th>
<th>SALES_95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Darjeeling Special</td>
<td>Bulk Tea</td>
<td>80610.50</td>
</tr>
<tr>
<td>Darjeeling Special</td>
<td>Pkg Tea</td>
<td>51266.00</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Bulk Tea</td>
<td>42329.00</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Bulk Tea</td>
<td>34592.75</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Bulk Tea</td>
<td>27763.75</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Pkg Tea</td>
<td>27192.50</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Bulk Tea</td>
<td>25848.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Bulk Tea</td>
<td>24594.00</td>
</tr>
<tr>
<td>Darjeeling Number 1</td>
<td>Pkg Tea</td>
<td>24232.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Bulk Tea</td>
<td>23269.50</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Bulk Tea</td>
<td>22326.00</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Bulk Tea</td>
<td>21964.00</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Bulk Tea</td>
<td>21584.50</td>
</tr>
<tr>
<td>Irish Breakfast</td>
<td>Pkg Tea</td>
<td>20084.00</td>
</tr>
<tr>
<td>English Breakfast</td>
<td>Pkg Tea</td>
<td>18955.00</td>
</tr>
<tr>
<td>Breakfast Blend</td>
<td>Pkg Tea</td>
<td>17031.50</td>
</tr>
<tr>
<td>Gold Tips</td>
<td>Pkg Tea</td>
<td>16783.25</td>
</tr>
<tr>
<td>Special Tips</td>
<td>Pkg Tea</td>
<td>16773.25</td>
</tr>
<tr>
<td>Assam Grade A</td>
<td>Pkg Tea</td>
<td>16724.00</td>
</tr>
<tr>
<td>Earl Grey</td>
<td>Pkg Tea</td>
<td>16108.00</td>
</tr>
</tbody>
</table>
Embedded Macros

An *embedded macro* is a macro that occurs within the definition of another macro.

**About the Example**

The CREATE MACRO statements in this example define two macros: *case_tea* and *tea_totals*.

- The first macro contains a CASE expression that replaces each Classkey with a meaningful class type—this is the same CASE expression that was used in the FROM clause of the query on page 6-2.
- The second macro is a complete SELECT statement that contains two embedded macros: *case_tea* and *tea_products* (defined on page 6-2).

To execute the macro *tea_totals*, just enter the macro name:

```
tea_totals;
```

The result set is identical to the one returned by the previous example on page 6-2.

**Usage Notes**

Macros can be embedded more than one level down.

A macro definition can contain the name of a defined macro, but it cannot contain another macro definition.
Macros with Parameters

Question

What were the total sales of tea products during any given year?

CREATE MACRO Statement

create macro tea_sales(yr) as
    select year, prod_name, case_tea,
    sum(dollars) as us_sales
    from product pt join sales sa
    on pt.classkey = sa.classkey and pt.prodkey = sa.prodkey
    join period pd on pd.perkey = sa.perkey
    where tea_products
    and year = yr
    group by year, prod_name, class
    order by us_sales desc;

Query

tea_sales(1994);

Result

<table>
<thead>
<tr>
<th>YEAR</th>
<th>PROD_NAME</th>
<th>CLASS</th>
<th>US_SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1994</td>
<td>Darjeeling Special</td>
<td>Bulk Tea</td>
<td>75582.00</td>
</tr>
<tr>
<td>1994</td>
<td>Darjeeling Special</td>
<td>Pkg Tea</td>
<td>51625.00</td>
</tr>
<tr>
<td>1994</td>
<td>Assam Gold Blend</td>
<td>Bulk Tea</td>
<td>43091.00</td>
</tr>
<tr>
<td>1994</td>
<td>Darjeeling Number 1</td>
<td>Bulk Tea</td>
<td>36442.00</td>
</tr>
<tr>
<td>1994</td>
<td>Assam Gold Blend</td>
<td>Pkg Tea</td>
<td>28328.00</td>
</tr>
<tr>
<td>1994</td>
<td>Irish Breakfast</td>
<td>Bulk Tea</td>
<td>27440.75</td>
</tr>
<tr>
<td>1994</td>
<td>English Breakfast</td>
<td>Bulk Tea</td>
<td>27071.00</td>
</tr>
<tr>
<td>1994</td>
<td>Darjeeling Number 1</td>
<td>Pkg Tea</td>
<td>25841.25</td>
</tr>
<tr>
<td>1994</td>
<td>Earl Grey</td>
<td>Bulk Tea</td>
<td>24721.00</td>
</tr>
<tr>
<td>1994</td>
<td>Breakfast Blend</td>
<td>Bulk Tea</td>
<td>24689.25</td>
</tr>
<tr>
<td>1994</td>
<td>Gold Tips</td>
<td>Bulk Tea</td>
<td>23181.25</td>
</tr>
<tr>
<td>1994</td>
<td>Special Tips</td>
<td>Bulk Tea</td>
<td>22712.25</td>
</tr>
<tr>
<td>1994</td>
<td>Assam Grade A</td>
<td>Bulk Tea</td>
<td>22418.00</td>
</tr>
<tr>
<td>1994</td>
<td>Irish Breakfast</td>
<td>Pkg Tea</td>
<td>21318.25</td>
</tr>
<tr>
<td>1994</td>
<td>Breakfast Blend</td>
<td>Pkg Tea</td>
<td>17606.25</td>
</tr>
<tr>
<td>1994</td>
<td>English Breakfast</td>
<td>Pkg Tea</td>
<td>17310.00</td>
</tr>
<tr>
<td>1994</td>
<td>Assam Grade A</td>
<td>Pkg Tea</td>
<td>16787.00</td>
</tr>
<tr>
<td>1994</td>
<td>Earl Grey</td>
<td>Pkg Tea</td>
<td>16416.00</td>
</tr>
<tr>
<td>1994</td>
<td>Special Tips</td>
<td>Pkg Tea</td>
<td>15883.75</td>
</tr>
<tr>
<td>1994</td>
<td>Gold Tips</td>
<td>Pkg Tea</td>
<td>15732.50</td>
</tr>
</tbody>
</table>
Macros, Views, and Temporary Tables

Macros with Parameters

A macro can be generalized with one or more parameters, which can be changed each time the macro is executed. For example, a macro can be written with a parameter for year so that the same macro retrieves values for any year stored in the database. Similarly, a macro that contains parameters for a product’s markets can retrieve data for any specified market.

CREATE MACRO Statement

Define a parameterized macro with the following command:

CREATE MACRO macro_name([parameter [, parameter] …]) AS definition;

where:

- **macro_name** A unique name that refers to the macro definition.
- **parameter** A value that customizes a generic macro. It can be changed each time the macro is used.
- **definition** A complete or partial SQL statement. Only one complete SQL statement can occur in the definition.

When you call a parameterized macro in a SELECT statement, you must include a value for each parameter defined in the CREATE MACRO statement.

About the Example

The CREATE MACRO statement defines a SELECT statement that contains a parameter for year (yr). When the macro `tea_sales(1994)` is executed, the warehouse server replaces each occurrence of the parameter `yr` with 1994. This query can be executed for any year for which sales data exists in the data warehouse (1994, 1995, or 1996 for the Aroma database).
Multi-Parameter Macros

Question

What were the best-selling products in a given location in a given year?

CREATE MACRO Statement

```
create macro top_rank(yr, locn, nbr) as
    select prod_name, city, year, sum(dollars) as sales,
           rank(sum(dollars)) as ranking
    from product pt join sales sa on pt.prodkey = sa.prodkey and pt.classkey = sa.classkey
          join period pd on pd.perkey = sa.perkey
          join store se on se.storekey = sa.storekey
    where city = locn and year = yr
    group by prod_name, city, year
    when rank(sum(dollars)) <= nbr;
```

Query 1 and Result

top_rank(1994, 'Los Angeles', 5);

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>CITY</th>
<th>YEAR</th>
<th>SALES</th>
<th>RANKING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xalapa Lapa</td>
<td>Los Angeles</td>
<td>1994</td>
<td>14930.00</td>
<td>1</td>
</tr>
<tr>
<td>Demitasse Ms</td>
<td>Los Angeles</td>
<td>1994</td>
<td>14402.25</td>
<td>2</td>
</tr>
<tr>
<td>Ruby's Allspice</td>
<td>Los Angeles</td>
<td>1994</td>
<td>14339.00</td>
<td>3</td>
</tr>
<tr>
<td>Aroma Roma</td>
<td>Los Angeles</td>
<td>1994</td>
<td>14253.25</td>
<td>4</td>
</tr>
<tr>
<td>Expresso XO</td>
<td>Los Angeles</td>
<td>1994</td>
<td>13179.50</td>
<td>5</td>
</tr>
</tbody>
</table>

Query 2 and Result

top_rank(1995, 'San Jose', 1);

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>CITY</th>
<th>YEAR</th>
<th>SALES</th>
<th>RANKING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demitasse Ms</td>
<td>San Jose</td>
<td>1995</td>
<td>32887.75</td>
<td>1</td>
</tr>
</tbody>
</table>

Query 3 and Result

top_rank(1996, 'Hartford', 3);

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>CITY</th>
<th>YEAR</th>
<th>SALES</th>
<th>RANKING</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA Lite</td>
<td>Hartford</td>
<td>1996</td>
<td>5061.00</td>
<td>1</td>
</tr>
<tr>
<td>Cafe Au Lait</td>
<td>Hartford</td>
<td>1996</td>
<td>4665.00</td>
<td>2</td>
</tr>
<tr>
<td>Xalapa Lapa</td>
<td>Hartford</td>
<td>1996</td>
<td>4610.00</td>
<td>3</td>
</tr>
</tbody>
</table>
**Macros with Multiple Parameters**

A macro can have multiple parameters. For example, you can define a macro with parameters for year, region, and product name.

**About the Example**

This CREATE MACRO statement defines a SELECT statement that contains three parameters:

\[(yr, locn, nbr)\]

which represent the year, the city (or location), and the maximum number of rank values to be returned.

Query 1 retrieves the five best-selling products in Los Angeles during 1994:

\[\text{top\_rank(1994, 'Los Ang\%', 5)};\]

When the macro is executed, the warehouse server replaces each occurrence of the three parameters with 1994, Los Angeles, and 5. Queries 2 and 3 show results for other years, locations, and rankings.

**Usage Notes**

Time periods such as days, weeks, months, quarters, and years are good candidates for parameters. So are product names, brands, trademarks, and suppliers.

The RANK function is discussed in detail in Chapter 3, “Data Analysis.”

There are two Aroma stores in San Jose, but only one in Hartford and Los Angeles; the results in Query 2 represent the sum of dollars for both San Jose stores.

The sales figures in the result set for Query 3 are significantly smaller because the Aroma database contains sales figures for only the first quarter of 1996 but for all four quarters of 1994 and 1995.
Macros, Views, and Temporary Tables

Comparisons

Question

How do the monthly sales of Lotta Latte in San Jose compare during the first quarters of 1995 and 1996, in terms of both dollars and quantities?

CREATE MACRO Statement

create macro lotta_sales(facts, yr) as (  
select sum(facts)  
from store t natural join sales s  
natural join product p  
natural join period d  
where d.month = e.month  
and d.year = e.yr  
and p.prod_name = q.prod_name  
and t.city = u.city);

Query 1 and Result

select q.prod_name, e.month, sum(dollars) as sales_95,  
lotta_sales(dollars, year+1) as sales_96  
from store u natural join product q natural join period e  
natural join sales l  
where qtr = 'Q1_95'  
and prod_name like 'Lotta Latte%'  
and city like 'San J%'  
group by q.prod_name, e.month, e.year, u.city;

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>MONTH</th>
<th>SALES_95</th>
<th>SALES_96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>JAN</td>
<td>1611.00</td>
<td>3475.00</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>FEB</td>
<td>3162.50</td>
<td>2409.50</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>MAR</td>
<td>2561.50</td>
<td>2831.50</td>
</tr>
</tbody>
</table>

Query 2 and Result

select q.prod_name, e.month, sum(dollars) as sales_95,  
lotta_sales(dollars, year+1) as sales_96,  
lotta_sales(quantity, year) as qty_95,  
lotta_sales(quantity, year+1) as qty_96  
from store u natural join product q natural join period e  
natural join sales l  
where qtr = 'Q1_95'  
and prod_name like 'Lotta Latte%'  
and city like 'San J%'  
group by q.prod_name, e.month, e.year, u.city;

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>MONTH</th>
<th>SALES_95</th>
<th>SALES_96</th>
<th>QTY_95</th>
<th>QTY_96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>JAN</td>
<td>1611.00</td>
<td>3475.00</td>
<td>197</td>
<td>426</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>FEB</td>
<td>3162.50</td>
<td>2409.50</td>
<td>391</td>
<td>298</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>MAR</td>
<td>2561.50</td>
<td>2831.50</td>
<td>314</td>
<td>348</td>
</tr>
</tbody>
</table>
Comparison Macros

A query is often built from a basic block of instructions that is repeated several times with minor variations. These variations are often good candidates for macro parameters. For example, a query that compares sales during the current year with sales during the previous year must contain similar blocks of instructions: One block retrieves sales for the current year, the other for the previous year. A macro that contains a parameter for “year” reduces the number of instructions you must enter manually.

About the Example

In examples 1 and 2, the main query retrieves monthly sales of Lotta Latte in San Jose during the first quarter of 1995 and the macro (a subquery) retrieves corresponding figures for 1996.

The macro subquery

```sql
lotta_sales(facts, yr)
```

can retrieve one of two types of facts—dollars or quantity—for a specified year. (The Sales table contains additive columns for dollars and quantity only; a production database would probably contain many more types of facts.)

In Query 1, the macro references the Dollars column of the Sales table and the year 1996:

```sql
lotta_sales(dollars, year+1)
```

The expression

```sql
year+1
```

evaluates to 1996 because the WHERE clause constraint in the main query refers to the year 1995.

This macro might not seem worth the effort to design until you begin to build more complex queries with it. For example, in Query 2, the macro is referenced three times, producing three different columns in the result set.

Usage Notes

This macro is based on a correlated subquery defined in the select list; however, equivalent subqueries defined in the FROM clause often result in faster performance, as described in Chapter 4.
Share Comparisons

Question

What were the monthly sales of Lotta Latte in San Jose during the first three months of 1996 and 1995? What was each month’s share (%) of the quarter in each year?

CREATE MACRO Statement

create macro lotta_qtr_sales(facts, yr) as
  (select sum(facts)
  from store t natural join sales s
      natural join product p
      natural join period d
  where substr(d.qtr,1,2) = substr(e.qtr,1,2)
      and d.year = e.yr
      and p.prod_name = q.prod_name
      and t.city = u.city);

Query

select q.prod_name, e.month, sum(dollars) as sales_95,
  dec(100*sales_95/lotta_qtr_sales(dollars, year),7,2) as share_qtr_95,
  lotta_sales(dollars, year+1) as sales_96,
  dec(100*sales_96/lotta_qtr_sales(dollars, year+1),7,2) as share_qtr_96
from store u natural join product q
    natural join period e
    natural join sales l
where qtr = 'Q1_95'
    and prod_name like 'Lotta Latte%'
    and city like 'San J%'
group by q.prod_name, e.month, e.qtr, e.year, u.city, sales_96;

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>MONTH</th>
<th>SALES_95</th>
<th>SHARE_QTR_95</th>
<th>SALES_96</th>
<th>SHARE_QTR_96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>JAN</td>
<td>1611.00</td>
<td>21.96</td>
<td>3475.00</td>
<td>39.86</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>FEB</td>
<td>3162.50</td>
<td>43.11</td>
<td>2409.50</td>
<td>27.64</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>MAR</td>
<td>2561.50</td>
<td>34.92</td>
<td>2831.50</td>
<td>32.48</td>
</tr>
</tbody>
</table>
Share Comparison Macros

Macros can also simplify calculations. For example, if you have a macro that retrieves monthly sales for a product and another macro that calculates the sum of that product’s sales during the quarter or year, you can easily calculate the monthly share of the total sales for the quarter or year.

This share is expressed as a simple percentage calculation. For example:

\[
100 \times \left( \frac{\text{monthly_sales}}{\text{quarterly_sales}} \right)
\]

This kind of macro can be applied to other years as well.

About the Example

This query retrieves monthly sales of Lotta Latte in San Jose during the first quarters of 1995 and 1996 and calculates each month’s share of the quarter during these years. This query would be much longer and more difficult to understand without the three macros.

The macro

\[
\text{lotta_sales(facts, yr)}
\]

works the same in this query as in the previous example in this chapter.

The macro

\[
\text{lotta_qtr_sales(facts, yr)}
\]

is another subquery that calculates, in this case, quarterly sales dollars for the specified year. (It could alternatively be used to calculate quarterly sales quantities.) The results of this macro are not displayed in the report but used as the source data for the two share calculations.

The SUBSTR function is used in the macro definition to correlate the Qtr column values based on their first two characters (Q1). This constraint is necessary because the Qtr values in the Period table are specific to each year (Q1_95 versus Q1_96, for example). For more information about the SUBSTR function, refer to the SQL Reference Guide.

Usage Notes

The GROUP BY clause must include the Sales_96 column, as well as the other non-aggregate columns that either appear in the select list or are referenced in the subqueries’ correlation conditions.
Change in Share

Question

When you compare the monthly sales of Lotta Latte in San Jose during the first quarter of 1995 and the first quarter of 1996:

- Did the sales figures for each month go up or down? By what percentage?
- Did the share-of-quarter percentage for each month go up or down? By what percentage?

Query

```sql
select q.prod_name, e.month, sum(dollars) as sales_95,
   lotta_sales(dollars, year+1) as sales_96,
   dec(100*(((sales_96 - sales_95)/sales_95),7,2) as sales_chg,
   dec(100*  
   ((sales_96/lotta_qtr_sales(dollars, year+1))  
   -  
   (sales_95/lotta_qtr_sales(dollars, year)))  
   ,7,2) as share_chg  
from store u natural join product q  
natural join period e natural join sales l  
where e.year = 1995  
and e.qtr = 'Q1_95'  
and q.prod_name like 'Lotta Latte%'  
and u.city like 'San J%'  
group by q.prod_name, e.month, e.qtr, e.year, u.city, sales_96;
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>MONTH</th>
<th>SALES_95</th>
<th>SALES_96</th>
<th>SALES_CHG</th>
<th>SHARE_CHG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotta Latte</td>
<td>JAN</td>
<td>1611.00</td>
<td>3475.00</td>
<td>115.70</td>
<td>17.90</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>FEB</td>
<td>3162.50</td>
<td>2409.50</td>
<td>-23.81</td>
<td>-15.47</td>
</tr>
<tr>
<td>Lotta Latte</td>
<td>MAR</td>
<td>2561.50</td>
<td>2831.50</td>
<td>10.54</td>
<td>-2.43</td>
</tr>
</tbody>
</table>
Macros That Calculate Change in Share

When analysts have macros at their disposal, they can think more about sales and markets and less about how to express their business questions with SQL.

For example, the change in a product’s monthly sales over two years can be expressed as a percentage by using the following calculation:

\[ 100 \times \frac{(\text{monthly\_sales\_96} - \text{monthly\_sales\_95})}{\text{monthly\_sales\_95}} \]

Similarly, the change in the product’s share of quarter can be calculated as

\[ 100 \times \left( \frac{\text{monthly\_sales\_96}}{\text{quarterly\_sales\_96}} \right) - \left( \frac{\text{monthly\_sales\_95}}{\text{quarterly\_sales\_95}} \right) \]

Neither percentage is difficult to calculate, but macros simplify the writing of a query that returns these percentages.

About the Example

The two previously defined macros (lotta_sales and lotta_qtr_sales) are used in this example to calculate the percentage change in monthly sales for the Lotta Latte product, as well as the change in share of quarter for the product’s sales in corresponding months from 1995 and 1996.
Views

Question

What were the sales totals and ranks by store for Assam Gold Blend tea in 1995?

CREATE VIEW Statement

```sql
create view tea_sales95
    as select prod_name, store_name, sum(dollars) as tea_dollars, rank(sum(dollars)) as tea_rank
    from sales natural join product
    natural join period
    natural join store
    where sales.classkey in (2, 5)
    and year = 1995
    group by prod_name, store_name;
```

Query

```sql
select prod_name, store_name, tea_dollars, tea_rank
from tea_sales95
where prod_name like 'Assam Gold%';
```

Result

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>STORE_NAME</th>
<th>TEA_DOLLARS</th>
<th>TEA_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assam Gold Blend</td>
<td>Beans of Boston</td>
<td>6201.50</td>
<td>1</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Beaches Brew</td>
<td>6080.00</td>
<td>2</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Texas Teahouse</td>
<td>5422.50</td>
<td>3</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Olympic Coffee Company</td>
<td>5350.50</td>
<td>4</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Cupertino Coffee Supply</td>
<td>5277.00</td>
<td>5</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Moroccan Moods</td>
<td>5178.50</td>
<td>6</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Coffee Brewers</td>
<td>5151.00</td>
<td>7</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Moulin Rouge Roasting</td>
<td>4977.00</td>
<td>8</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>East Coast Roast</td>
<td>4769.00</td>
<td>9</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Miami Espresso</td>
<td>4506.50</td>
<td>10</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Roasters, Los Gatos</td>
<td>4414.50</td>
<td>11</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>San Jose Roasting Company</td>
<td>4226.50</td>
<td>12</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Instant Coffee</td>
<td>4190.50</td>
<td>13</td>
</tr>
<tr>
<td>Assam Gold Blend</td>
<td>Java Judy's</td>
<td>3776.50</td>
<td>14</td>
</tr>
</tbody>
</table>
Selecting from Views

Analysts might be interested in certain products or time periods only, rather than the full range of facts and dimensions stored in the data warehouse. You can create views—read-only tables that contain subsets of information from existing tables or views—to make access to the specific data you want to query both easier and faster.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS query_expression
```

where `query_expression` is any join or non-join query expression, as defined in Chapter 7 of the SQL Reference Guide.

About the Example

This view contains four columns:

- Product names (`Prod_Name`)
- Store names (`Store_Name`)
- Aggregated sales totals per store, per tea product for 1995 (`Tea_Totals`)
- Rankings based on the aggregated sales totals (`Tea_Rank`)

The query simply constrains on the `Prod_Name` column to return sales totals and ranks per store for Assam Gold Blend tea.

The search condition

```
where sales.classkey in (2, 5)
```

ensures that only tea products are selected by the view. (In the Class table, the Classkey values are meaningful and map to specific groups of products.)

Usage Notes

This business question can be asked with or without creating the view; however, the view improves performance and simplifies the analyst’s approach to writing queries.

Query expressions cannot contain ORDER BY clauses; therefore, it might not be practical to include an order-dependent display function in a CREATE VIEW statement. Because the RANK function is not order-dependent (unlike CUME, for example), it is used successfully in this example. For detailed information about RISQL display functions, refer to the SQL Reference Guide.
**Macros, Views, and Temporary Tables**

**INSERT INTO SELECT Statement**

**Question**

Create a temporary table to hold daily and cumulative sales totals for clothing products. Issue a SELECT statement on the table to retrieve only data for stores in Los Angeles.

**CREATE TEMPORARY TABLE Statement**

```sql
create temporary table clothing_sales
(date date,
prod_name char(30),
city char(20),
dollars dec(7,2),
cume_tot integer);
```

**INSERT Statement**

```sql
insert into clothing_sales
(date, prod_name, city, dollars, cume_tot)
select date, prod_name, city, dollars, cume(dollars)
from store s
join sales l on s.storekey = l.storekey
join period t on l.perkey = t.perkey
join product p on l.classkey = p.classkey
and l.prodkey = p.prodkey
join class c on p.classkey = c.classkey
where class_type = 'Clothing'
order by date, city
reset by date;
```

**INFORMATION** (209) Rows inserted: 816.

**Query**

```sql
select date, prod_name, dollars, cume_tot
from clothing_sales
where city = 'Los Angeles'
and extract(year from date) = 1996
order by date;
```

**Result**

<table>
<thead>
<tr>
<th>DATE</th>
<th>PROD_NAME</th>
<th>DOLLARS</th>
<th>CUME_TOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996-01-08</td>
<td>Aroma t-shirt</td>
<td>197.10</td>
<td>308</td>
</tr>
<tr>
<td>1996-01-18</td>
<td>Aroma t-shirt</td>
<td>131.40</td>
<td>131</td>
</tr>
<tr>
<td>1996-01-18</td>
<td>Aroma baseball cap</td>
<td>135.15</td>
<td>266</td>
</tr>
<tr>
<td>1996-01-23</td>
<td>Aroma baseball cap</td>
<td>15.90</td>
<td>15</td>
</tr>
<tr>
<td>1996-02-01</td>
<td>Aroma t-shirt</td>
<td>175.20</td>
<td>175</td>
</tr>
<tr>
<td>1996-02-04</td>
<td>Aroma t-shirt</td>
<td>164.25</td>
<td>164</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Creating a Temporary Table

If you have resource or DBA authorization for a database, you can create a temporary table that contains the result set of a query. Temporary tables are useful when you want to perform repeated analysis on a result set without reprocessing the original query. For example, you can store the results of RISQL display functions in temporary tables and issue SELECT statements against them to further constrain the result data. When you use display functions to fill a temporary table, remember to order the data to ensure that the results of the display-function column are accurate.

**INSERT INTO SELECT**

```
INSERT INTO table_name select_statement
```

where `table_name` is a valid table name and `select_statement` is a complete or partial SELECT statement, as defined in Chapter 7 of the SQL Reference Guide.

**About the Example**

This example shows how to create a temporary table named Clothing_Sales, insert daily and cumulative sales totals into it, and query it by issuing a standard SELECT statement.

The results of the query in the example could be retrieved with more limiting search conditions in a regular SELECT statement. However, creating a temporary table to store cumulative totals improves query performance when you are working with large fact tables.

**Usage Notes**

Temporary tables are removed from the database automatically when your SQL session ends. These tables are not visible to other users connected to the same database.

To create tables in a warehouse database, you must have resource or DBA authorization. Along with resource and DBA authorization comes INSERT privilege, which allows you to insert data into any tables you create. For a complete discussion of authorizations and privileges, refer to the SQL Reference Guide.

The CREATE TABLE statement for a temporary table must define columns that are of the same datatype and size as columns defined in the base tables of the database. Otherwise, input data from the INSERT INTO...SELECT statement will be incompatible with columns in the temporary table.
Macros, Views, and Temporary Tables

Summary

This chapter showed how to simplify SQL statements with RISQL macros, and how to create views and temporary tables by using the CREATE VIEW, CREATE TEMPORARY TABLE, and INSERT INTO...SELECT commands.

CREATE MACRO Statement

CREATE MACRO macro_name(parameter [, parameter] ... ) AS definition ;

A macro name is a character string that begins with a letter and does not exceed 128 characters. Macro names are not case-sensitive. A RISQL keyword cannot be a macro name.

When you call a parameterized macro, you must include a value for each parameter defined in the CREATE MACRO statement.

CREATE VIEW Statement

CREATE VIEW view_name AS query_expression

CREATE TEMPORARY TABLE Statement

CREATE TEMPORARY TABLE table_name (column_definitions)

INSERT INTO SELECT Statement

INSERT INTO table_name select_statement
Appendix A describes all of the tables in the Aroma database, which consists of two schemas—a simple star schema for retail sales information and a multi-star schema for purchasing information.

Most of the examples in this document use the tables in the retail schema. The purchasing tables are used in a few examples that require a more flexible schema for adequate illustration.
Aroma Database—Retail Schema

Most of the examples in this guide are based on data from the basic Aroma database, which tracks daily retail sales in stores owned by the Aroma Coffee and Tea Company. The following figure illustrates this basic schema.

The crow’s-feet in this diagram indicate a one-to-many relationship between the two tables. For example, each distinct value in the Perkey column of the Period table can occur only once in that table but many times in the Sales table.
Basic Aroma Schema

The following tables comprise the basic Aroma database:

**Period**
Defines time intervals such as days, months, and years.

**Class**
Defines classes of products sold at retail stores.

**Product**
Defines individual products sold at retail stores, including bulk and packaged coffee and tea, coffee machines, and so on.

**Market**
Defines the geographical markets of the business.

**Store**
Defines individual retail stores owned and operated by the Aroma Coffee and Tea Company.

**Promotion**
Defines the types, durations, and values of promotions run on different products.

**Sales**
Contains the sales figures for Aroma products during time periods at various stores.

The Period, Class, Product, Market, Store, and Promotion tables are examples of typical business dimensions: They are small and contain descriptive data that is familiar to users.

The Sales table is a good example of a fact table: It contains thousands of rows, and its largely additive information is accessed in queries via joins to the dimension tables it references.
Query

select * from class;

Result

<table>
<thead>
<tr>
<th>CLASSKEY</th>
<th>CLASS_TYPE</th>
<th>CLASS_DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bulk_beans</td>
<td>Bulk coffee products</td>
</tr>
<tr>
<td>2</td>
<td>Bulk_tea</td>
<td>Bulk tea products</td>
</tr>
<tr>
<td>3</td>
<td>Bulk_spice</td>
<td>Bulk spices</td>
</tr>
<tr>
<td>4</td>
<td>Pkg_coffee</td>
<td>Individually packaged coffee products</td>
</tr>
<tr>
<td>5</td>
<td>Pkg_tea</td>
<td>Individually packaged tea products</td>
</tr>
<tr>
<td>6</td>
<td>Pkg_spice</td>
<td>Individually packaged spice products</td>
</tr>
<tr>
<td>7</td>
<td>Hardware</td>
<td>Coffee mugs, teapots, spice jars, espresso machines</td>
</tr>
<tr>
<td>8</td>
<td>Gifts</td>
<td>Samplers, gift boxes and baskets, etc.</td>
</tr>
<tr>
<td>12</td>
<td>Clothing</td>
<td>T-shirts, caps, etc.</td>
</tr>
</tbody>
</table>

Query

select * from product;

Result

<table>
<thead>
<tr>
<th>CLASSKEY</th>
<th>PRODKEY</th>
<th>PROD_NAME</th>
<th>PKG_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Veracruzano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Xalapa Lapa</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Colombiano</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Expresso XO</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>La Antigua</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>Lotta Latte</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>Cafe Au Lait</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>NA Lite</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>Aroma Roma</td>
<td>No pkg</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>Demitasse Ms</td>
<td>No pkg</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Darjeeling Number 1</td>
<td>No pkg</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Darjeeling Special</td>
<td>No pkg</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Assam Grade A</td>
<td>No pkg</td>
</tr>
</tbody>
</table>
The Class and Product Tables

The Product table describes the products defined in the Aroma database. The Class table describes the classes to which those products belong.

If a dimension table contains foreign key columns that reference other dimension tables, the referenced tables are called outboard or outrigger tables. The Product table’s Classkey column is a foreign-key reference to the Class table, so the Class table is an outboard table.

Class Table—Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>classkey</td>
<td>Integer that identifies exactly one row in the Class table. Classkey is the primary key.</td>
</tr>
<tr>
<td>class_type</td>
<td>Character string that identifies a group of products.</td>
</tr>
<tr>
<td>class_desc</td>
<td>Character string that describes a group of products.</td>
</tr>
</tbody>
</table>

Product Table—Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>classkey</td>
<td>Foreign-key reference to the Class table.</td>
</tr>
<tr>
<td>prodkey</td>
<td>Integer that combines with the Classkey value to identify exactly one row in the Product table. Classkey/Prodkey is a two-column primary key.</td>
</tr>
<tr>
<td>prod_name</td>
<td>Character string that identifies a product. The database contains 59 products. A fully populated database would have many more. (Note that some Aroma products have the same name but belong to different classes and have different package types.)</td>
</tr>
<tr>
<td>pkg_type</td>
<td>Character string that identifies the type of packaging for each product.</td>
</tr>
</tbody>
</table>
### Sample Data from the Store and Market Tables

#### Query

```sql
select * from market;
```

#### Result

<table>
<thead>
<tr>
<th>MKTKEY</th>
<th>HQ_CITY</th>
<th>HQ_STATE</th>
<th>DISTRICT</th>
<th>REGION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Atlanta</td>
<td>GA</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>2</td>
<td>Miami</td>
<td>FL</td>
<td>Atlanta</td>
<td>South</td>
</tr>
<tr>
<td>3</td>
<td>New Orleans</td>
<td>LA</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>4</td>
<td>Houston</td>
<td>TX</td>
<td>New Orleans</td>
<td>South</td>
</tr>
<tr>
<td>5</td>
<td>New York</td>
<td>NY</td>
<td>New York</td>
<td>North</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Query

```sql
select * from store;
```

#### Result

<table>
<thead>
<tr>
<th>STOREKEY</th>
<th>MKTKEY</th>
<th>STORE_TYPE</th>
<th>STORE_NAME</th>
<th>STREET</th>
<th>CITY</th>
<th>STATE</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>Small</td>
<td>Roasters, Los Gatos</td>
<td>1234 University A</td>
<td>Los Gatos</td>
<td>CA</td>
<td>95032</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>Large</td>
<td>San Jose Roasting Co</td>
<td>5678 Bascom Ave</td>
<td>San Jose</td>
<td>CA</td>
<td>95156</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>Medium</td>
<td>Cupertino Coffee Sup</td>
<td>987 DeAnza Blvd</td>
<td>Cupertino</td>
<td>CA</td>
<td>97865</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Medium</td>
<td>Moulin Rouge Roastin</td>
<td>898 Main Street</td>
<td>New Orleans</td>
<td>LA</td>
<td>70125</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>Small</td>
<td>Moon Pennies</td>
<td>98675 University</td>
<td>Detroit</td>
<td>MI</td>
<td>48209</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>Small</td>
<td>The Coffee Club</td>
<td>9865 Lakeshore Bl</td>
<td>Chicago</td>
<td>IL</td>
<td>60606</td>
</tr>
</tbody>
</table>

...
The Market and Store Tables

The Store table defines the stores that sell Aroma products. The Market table describes the U.S. markets to which each store belongs. Each market is identified by a major metropolitan city. The Market table is an outboard table, like the Class table.

Market Table—Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>mktkey</td>
<td>Integer that identifies exactly one row in the Market table. Mktkey is the primary key.</td>
</tr>
<tr>
<td>hq_city</td>
<td>Character string that identifies a city. The Market table defines 17 cities. A fully populated database could have thousands.</td>
</tr>
<tr>
<td>state</td>
<td>Character string that identifies a state.</td>
</tr>
<tr>
<td>district</td>
<td>Character string that identifies a district based on a major metropolitan city. A global database would contain countries and nations or other geographic dimensions.</td>
</tr>
<tr>
<td>region</td>
<td>Character string that identifies a region. The Market table defines only 4 regions for the entire United States. A comprehensive database would include numerous regions and probably more districts within a region.</td>
</tr>
</tbody>
</table>

Store Table—Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>storekey</td>
<td>Integer that identifies exactly one row in the Store table. Storekey is the primary key.</td>
</tr>
<tr>
<td>mktkey</td>
<td>Foreign-key reference to the Market table.</td>
</tr>
<tr>
<td>store_type</td>
<td>Character string that identifies stores by size.</td>
</tr>
<tr>
<td>store_name</td>
<td>Character string that identifies a store by name.</td>
</tr>
<tr>
<td>street, city, state, zip</td>
<td>Columns that identify each store’s address.</td>
</tr>
</tbody>
</table>
The Complete Aroma Database

Sample Data from the Period Table

Query

```sql
select * from period;
```

Result

```
PERKEY DATE      DAY  WEEK MONTH QTR   YEAR
1 1994-01-01 SA  1 JAN Q1_94 1994
2 1994-01-02 SU  2 JAN Q1_94 1994
3 1994-01-03 MO  2 JAN Q1_94 1994
4 1994-01-04 TU  2 JAN Q1_94 1994
5 1994-01-05 WE  2 JAN Q1_94 1994
6 1994-01-06 TH  2 JAN Q1_94 1994
7 1994-01-07 FR  2 JAN Q1_94 1994
8 1994-01-08 SA  2 JAN Q1_94 1994
9 1994-01-09 SU  3 JAN Q1_94 1994
10 1994-01-10 MO 3 JAN Q1_94 1994
11 1994-01-11 TU 3 JAN Q1_94 1994
12 1994-01-12 WE 3 JAN Q1_94 1994
13 1994-01-13 TH 3 JAN Q1_94 1994
14 1994-01-14 FR 3 JAN Q1_94 1994
15 1994-01-15 SA 3 JAN Q1_94 1994
16 1994-01-16 SU 4 JAN Q1_94 1994
17 1994-01-17 MO 4 JAN Q1_94 1994
18 1994-01-18 TU 4 JAN Q1_94 1994
19 1994-01-19 WE 4 JAN Q1_94 1994
20 1994-01-20 TH 4 JAN Q1_94 1994
...
The Period table defines daily, weekly, monthly, quarterly, and yearly time periods for 1994 and 1995 and the first quarter of 1996.

**Column Descriptions**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>perkey</td>
<td>Integer that identifies exactly one row in the Period table. Perkey is the primary key.</td>
</tr>
<tr>
<td>date</td>
<td>Date value that identifies each day from January 1, 1994 through March 31, 1996.</td>
</tr>
<tr>
<td>day</td>
<td>Character-string abbreviation of the day of the week.</td>
</tr>
<tr>
<td>week</td>
<td>Integer that identifies each week of each year by number (1 through 53, each new week starting on a Sunday).</td>
</tr>
<tr>
<td>month</td>
<td>Character-string abbreviation of the name of each month.</td>
</tr>
<tr>
<td>qtr</td>
<td>Character string that uniquely identifies each quarter (for example, Q1_94, Q3_95).</td>
</tr>
<tr>
<td>year</td>
<td>Integer that identifies the year.</td>
</tr>
</tbody>
</table>
**The Complete Aroma Database**

**Sample Data from the Promotion Table**

**Query**

```sql
select * from promotion;
```

**Result**

<table>
<thead>
<tr>
<th>PROMOKEY</th>
<th>PROMO_TYPE</th>
<th>PROMO_DESC</th>
<th>VALUE</th>
<th>START_DATE</th>
<th>END_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>No promotion</td>
<td>0.00</td>
<td>9999-01-01</td>
<td>9999-01-01</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-01-01</td>
<td>1994-01-31</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-02-01</td>
<td>1994-02-28</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-03-01</td>
<td>1994-03-31</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-04-01</td>
<td>1994-04-30</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-05-01</td>
<td>1994-05-31</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-06-01</td>
<td>1994-06-30</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-07-01</td>
<td>1994-07-31</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-08-01</td>
<td>1994-08-31</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-09-01</td>
<td>1994-09-30</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-10-01</td>
<td>1994-10-31</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-11-01</td>
<td>1994-11-30</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1994-12-01</td>
<td>1994-12-31</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1995-01-01</td>
<td>1995-01-31</td>
</tr>
<tr>
<td>14</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1995-02-01</td>
<td>1995-02-28</td>
</tr>
<tr>
<td>15</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1995-03-01</td>
<td>1995-03-31</td>
</tr>
<tr>
<td>16</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1995-04-01</td>
<td>1995-04-30</td>
</tr>
<tr>
<td>17</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1995-05-01</td>
<td>1995-05-31</td>
</tr>
<tr>
<td>18</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1995-06-01</td>
<td>1995-06-30</td>
</tr>
<tr>
<td>19</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1995-07-01</td>
<td>1995-07-31</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>Aroma catalog coupon</td>
<td>1.00</td>
<td>1995-08-01</td>
<td>1995-08-31</td>
</tr>
</tbody>
</table>

...
The Promotion Table

The Promotion table is a dimension table that describes promotions that are run on different products during different time periods. Promotion tables are sometimes referred to as condition tables because they indicate the conditions under which goods are sold.

Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>promokey</td>
<td>Integer that identifies exactly one row in the Promotion table. Promokey is the primary key.</td>
</tr>
<tr>
<td>promo_type</td>
<td>Integer that identifies the promotion by number (or code).</td>
</tr>
<tr>
<td>promo_desc</td>
<td>Character string that describes the promotion type.</td>
</tr>
<tr>
<td>value</td>
<td>Decimal number that represents the dollar value of the promotion, such as a price reduction or the value of a coupon.</td>
</tr>
<tr>
<td>start_date,</td>
<td>Date values that indicate when each promotion begins and ends.</td>
</tr>
<tr>
<td>end_date</td>
<td></td>
</tr>
</tbody>
</table>
The Complete Aroma Database

Sample Data from the Sales Table

Query

select * from sales;

Result

<table>
<thead>
<tr>
<th>PERKEY</th>
<th>CLASSKEY</th>
<th>PRODKEY</th>
<th>STOREKEY</th>
<th>PROMOKEY</th>
<th>QUANTITY</th>
<th>DOLLARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>116</td>
<td>8</td>
<td>34.00</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>116</td>
<td>9</td>
<td>60.75</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>116</td>
<td>40</td>
<td>270.00</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>30</td>
<td>1</td>
<td>116</td>
<td>16</td>
<td>36.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>22</td>
<td>1</td>
<td>116</td>
<td>11</td>
<td>30.25</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>30</td>
<td>1</td>
<td>116</td>
<td>30</td>
<td>187.50</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>116</td>
<td>25</td>
<td>143.75</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>12</td>
<td>87.00</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>115.50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>22</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>58.50</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>17</td>
<td>136.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>74.75</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>30</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>101.50</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>63.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>22</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>99.00</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>46</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>36.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>10</td>
<td>40.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>36</td>
<td>279.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>132.00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td>48.00</td>
</tr>
</tbody>
</table>

...
The Sales Table

The Sales table is a fact table; as such, it is by far the largest table in the Aroma database and its data is stored in two distinct areas of the database (known as segments). For information about segments, refer to the Warehouse Administrator's Guide. The Sales table is large compared with Aroma’s other tables, but small compared with typical fact tables at customer sites, which usually contain millions of rows.

Multi-Part Primary Key

The Sales table contains a multi-part primary key: Each of its five columns is a foreign key reference to another table’s primary key:

perkey, classkey, prodkey, storekey, promokey

This primary key links the Sales data to the Period, Product, Store, and Promotion dimensions.

To improve query performance, a STARindex™ structure is built on the composite primary key of the Sales table. The presence of the STAR index makes STARjoin™ processing possible when the retail tables are joined in queries. For detailed examples of queries that require joins, refer to Chapter 5, “Joins and Unions.” For detailed information about STAR indexes, refer to the Warehouse Administrator’s Guide.

Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>perkey</td>
<td>Foreign-key reference to the Period table.</td>
</tr>
<tr>
<td>classkey</td>
<td>Foreign-key reference to the Product table.</td>
</tr>
<tr>
<td>prodkey</td>
<td>Foreign-key reference to the Product table.</td>
</tr>
<tr>
<td>storekey</td>
<td>Foreign-key reference to the Store table.</td>
</tr>
<tr>
<td>promokey</td>
<td>Foreign-key reference to the Promotion table.</td>
</tr>
<tr>
<td>quantity</td>
<td>Integer that represents the total quantity sold (per day).</td>
</tr>
<tr>
<td>dollars</td>
<td>Decimal number that represents dollar sales figures (per day).</td>
</tr>
</tbody>
</table>
Aroma Database—Purchasing Schema

A few of the examples in this guide are based on tables used to track product orders that the Aroma Company receives from its suppliers. This purchasing schema uses the same Product, Class, and Period dimensions as the retail schema, but has two dimensions of its own—Deal and Supplier. The Line_Items and Orders tables both contain facts, but the Orders table can also be queried as a dimension table referenced by the Line_Items table.

The following figure illustrates the tables in the purchasing schema.

Multi-Star Schema

The primary keys of the Line_Items and Orders tables do not match the set of their respective dimension table foreign keys. Any given combination of dimension table primary keys can point to more than one row in these fact tables; this type of table is known as a multi-star fact table or data list.

For example, multiple order numbers in the Orders table can refer to the same set of Supplier, Deal, and Period characteristics:

<table>
<thead>
<tr>
<th>ORDER_NO</th>
<th>PERKEY</th>
<th>SUPKEY</th>
<th>DEALKEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3699</td>
<td>817</td>
<td>1007</td>
<td>0</td>
</tr>
<tr>
<td>3700</td>
<td>817</td>
<td>1007</td>
<td>0</td>
</tr>
</tbody>
</table>
The purchasing schema contains similar kinds of facts to those stored in the Sales table—prices and quantities. The prices are dollar values representing amounts paid to suppliers for whole orders or specific line items within orders. The quantities represent units of product ordered.

You can use this schema to ask interesting questions about Aroma’s purchasing history—which suppliers give the best deal on which products, or which suppliers have the best record for closing orders, for example.

The Aroma Company sells the same products throughout its stores as it orders through its suppliers; therefore you can write queries that span both schemas to compare what was ordered with what was sold or to calculate simple profit margins.

The following tables comprise the purchasing schema of the Aroma database:

- **Period**: Defines time intervals such as days, months, and years.
- **Class**: Defines classes of products (both sold at retail stores and ordered from suppliers).
- **Product**: Defines individual products (both sold at retail stores and ordered from suppliers).
- **Supplier**: Defines the suppliers of products ordered by the Aroma Company.
- **Deal**: Defines the discount deals applied to orders by suppliers.
- **Line_Items**: Contains the line-item detail information for product orders, including the price and quantity of each item on each order.
- **Orders**: Contains information about product orders, such as the full price of each order, the types of products ordered, and so on.

The Supplier and Deal tables are exclusive to the purchasing schema and are referenced by the Orders table.

**Note**: The purchasing schema contains data for the first quarter of 1996 only.
Sample Data from the Supplier and Deal Tables

Query

```sql
select * from supplier;
```

Result

```
SUPKEY   TYPE         NAME         STREET          CITY            STATE ZIP
1001    Bulk coffee  CB Imports   100 Church Stre Mountain View CA    94001
1002    Bulk tea     Tea Makers,  1555 Hicks Rd.  San Jose        CA    95124
...
```

Some columns have been truncated to fit on the page.

Query

```sql
select * from deal;
```

Result

```
DEALKEY     DEAL_TYPE   DEAL_DESC                                DISCOUNT
0        1000 No deal                                       0.00
1         100 Orders over $10,000                         100.00
2         100 Orders over $20,000                         500.00
3         100 Supplier catalog coupon                      50.00
4         100 Supplier catalog coupon                     100.00
37        200 Supplier coffee special                      75.00
38        200 Supplier coffee special                     50.00
39        200 Supplier tea special                         40.00
40        200 Supplier tea special                         20.00
```
The Supplier and Deal Tables

Supplier Table—Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>supkey</td>
<td>Integer that identifies exactly one row in the Supplier table. Supkey is the primary key.</td>
</tr>
<tr>
<td>type</td>
<td>Character string that indicates the type of products supplied.</td>
</tr>
<tr>
<td>name</td>
<td>Character string that identifies the supplier by name.</td>
</tr>
<tr>
<td>street, city, state, zip</td>
<td>Columns that identify the supplier’s address.</td>
</tr>
</tbody>
</table>

Deal Table—Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>dealkey</td>
<td>Integer that identifies exactly one row in the Deal table. Dealkey is the primary key.</td>
</tr>
<tr>
<td>deal_type</td>
<td>Integer that identifies the type of deal (a code number).</td>
</tr>
<tr>
<td>deal_desc</td>
<td>Character string that describes the type of deal.</td>
</tr>
<tr>
<td>discount</td>
<td>Decimal value that indicates the dollar amount of the deal applied to an order.</td>
</tr>
</tbody>
</table>

Shared Dimensions

The purchasing schema shares the Period, Product, and Class tables with the retail schema.

As well as querying the retail and purchasing schemas independently, you can pose some interesting questions that involve tables from both schemas. For example, you can join the Sales and Line_Items tables to compare quantities of products ordered with quantities of products sold. A query like this uses the shared dimensions to constrain on products and periods.
### Sample Data from the Orders and Line_Items Tables

#### Query

```sql
select * from orders;
```

#### Result

<table>
<thead>
<tr>
<th>ORDER_NO</th>
<th>PERKEY</th>
<th>SUPKEY</th>
<th>DEALKEY</th>
<th>ORDER_TYPE</th>
<th>ORDER_DESC</th>
<th>CLOSE_DATE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3600</td>
<td>731</td>
<td>1001</td>
<td>37</td>
<td>Coffee</td>
<td>Whole coffee b</td>
<td>1996-01-07</td>
<td>1200.46</td>
</tr>
<tr>
<td>3601</td>
<td>732</td>
<td>1001</td>
<td>37</td>
<td>Coffee</td>
<td>Whole coffee b</td>
<td>1996-01-07</td>
<td>1535.94</td>
</tr>
<tr>
<td>3602</td>
<td>733</td>
<td>1001</td>
<td>0</td>
<td>Tea</td>
<td>Loose tea, bul</td>
<td>1996-01-07</td>
<td>780.00</td>
</tr>
<tr>
<td>3603</td>
<td>740</td>
<td>1001</td>
<td>39</td>
<td>Tea</td>
<td>Loose tea, bul</td>
<td>1996-01-21</td>
<td>956.45</td>
</tr>
<tr>
<td>3604</td>
<td>744</td>
<td>1005</td>
<td>0</td>
<td>Spice</td>
<td>Pre-packed spi</td>
<td>1996-01-16</td>
<td>800.66</td>
</tr>
<tr>
<td>3605</td>
<td>768</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>1996-02-12</td>
<td>25100.00</td>
</tr>
<tr>
<td>3606</td>
<td>775</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>1996-02-19</td>
<td>25100.00</td>
</tr>
<tr>
<td>3607</td>
<td>782</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>1996-02-25</td>
<td>25100.00</td>
</tr>
<tr>
<td>3608</td>
<td>789</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>1996-03-03</td>
<td>30250.00</td>
</tr>
<tr>
<td>3609</td>
<td>796</td>
<td>1003</td>
<td>2</td>
<td>Coffee</td>
<td>Whole-bean and</td>
<td>1996-03-15</td>
<td>25100.00</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Query

```sql
select * from line_items;
```

#### Result

<table>
<thead>
<tr>
<th>ORDER_NO</th>
<th>LINE_ITEM</th>
<th>PERKEY</th>
<th>CLASSKEY</th>
<th>PRODKEY</th>
<th>RECEIVE_DA</th>
<th>QUANTITY</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3600</td>
<td>1</td>
<td>731</td>
<td>1</td>
<td>1</td>
<td>1996-01-07</td>
<td>40</td>
<td>180.46</td>
</tr>
<tr>
<td>3600</td>
<td>2</td>
<td>731</td>
<td>2</td>
<td>10</td>
<td>1996-01-07</td>
<td>150</td>
<td>300.00</td>
</tr>
<tr>
<td>3600</td>
<td>3</td>
<td>731</td>
<td>2</td>
<td>11</td>
<td>1996-01-07</td>
<td>80</td>
<td>240.00</td>
</tr>
<tr>
<td>3600</td>
<td>4</td>
<td>731</td>
<td>2</td>
<td>12</td>
<td>1996-01-07</td>
<td>150</td>
<td>240.00</td>
</tr>
<tr>
<td>3600</td>
<td>5</td>
<td>731</td>
<td>1</td>
<td>20</td>
<td>1996-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>1</td>
<td>732</td>
<td>1</td>
<td>0</td>
<td>1996-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>2</td>
<td>732</td>
<td>1</td>
<td>1</td>
<td>1996-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>3</td>
<td>732</td>
<td>1</td>
<td>10</td>
<td>1996-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>4</td>
<td>732</td>
<td>1</td>
<td>11</td>
<td>1996-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>5</td>
<td>732</td>
<td>1</td>
<td>12</td>
<td>1996-01-07</td>
<td>60</td>
<td>240.00</td>
</tr>
<tr>
<td>3601</td>
<td>6</td>
<td>732</td>
<td>1</td>
<td>31</td>
<td>1996-01-07</td>
<td>70</td>
<td>335.94</td>
</tr>
<tr>
<td>3602</td>
<td>1</td>
<td>733</td>
<td>2</td>
<td>0</td>
<td>1996-01-08</td>
<td>70</td>
<td>130.00</td>
</tr>
<tr>
<td>3602</td>
<td>2</td>
<td>733</td>
<td>2</td>
<td>1</td>
<td>1996-01-08</td>
<td>70</td>
<td>130.00</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Orders and Line_Items Tables

The Orders and Line_Items tables contain the purchasing facts. For more details about these tables, see page A-14.

Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>order_no</td>
<td>Integer that identifies exactly one row in the Orders table. Order_No is the primary key.</td>
</tr>
<tr>
<td>perkey</td>
<td>Foreign-key reference to the Period table.</td>
</tr>
<tr>
<td>supkey</td>
<td>Foreign-key reference to the Supplier table.</td>
</tr>
<tr>
<td>dealkey</td>
<td>Foreign-key reference to the Deal table.</td>
</tr>
<tr>
<td>order_type</td>
<td>Character string that defines the types of products on the order.</td>
</tr>
<tr>
<td>order_desc</td>
<td>Character string that describes the type of order.</td>
</tr>
<tr>
<td>close_date</td>
<td>Date value that identifies when the order was completed or closed.</td>
</tr>
<tr>
<td>price</td>
<td>Decimal value that indicates the full cost of the order.</td>
</tr>
</tbody>
</table>

Column Descriptions

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>order_no</td>
<td>Integer that identifies exactly one row in the Orders table. Order_No is the primary key.</td>
</tr>
<tr>
<td>line_item</td>
<td>Integer that identifies each item listed on the order by number.</td>
</tr>
<tr>
<td>perkey</td>
<td>Foreign-key reference to the Period table.</td>
</tr>
<tr>
<td>classkey</td>
<td>Foreign-key reference to the Product table.</td>
</tr>
<tr>
<td>prodkey</td>
<td>Foreign-key reference to the Product table.</td>
</tr>
<tr>
<td>receive_date</td>
<td>Date value that identifies when the line-item was received.</td>
</tr>
<tr>
<td>quantity</td>
<td>Integer that indicates the quantity of products ordered for each line-item.</td>
</tr>
<tr>
<td>price</td>
<td>Decimal value that indicates the cost of the line-item.</td>
</tr>
</tbody>
</table>
The Complete Aroma Database
Symbols
% , SQL wildcard character 2-17
( ), for order of precedence 2-33
+ , arithmetic operator 2-33
- , arithmetic operator 2-33
_ , SQL wildcard character 2-17

A
addition operator 2-33
aggregate queries 2-29
aggregate tables
not in Aroma database 1-3
queries that require 3-3
aggregation functions 2-25
aliases, column 2-21, 2-27
ALL comparison predicate 2-15, 4-21
AND connective 2-9
ANY comparison predicate 2-15, 4-25
arithmetic operators, listed 2-33
Aroma database 1-2, A-2
Class table 1-4, A-4
column descriptions A-5 to A-19
Deal table A-16
dimension tables
purchasing schema A-15
retail schema 1-4, A-4
fact tables
purchasing schema A-19
retail schema 1-6, A-12
illustrated
purchasing schema A-14
retail schema 1-2, A-2
Line_Items table A-18
Market table 1-5, A-6
Orders table A-18
Period table 1-4, A-8

Aroma database, continued
Product table 1-4, A-4
Promotion table 1-5, A-10
purchasing schema A-14
retail schema 1-2, A-2
Sales table 1-7, A-12
Store table 1-5, A-6
Supplier table A-16
AS keyword, for column aliases 2-27
ASC keyword 2-21
ascending order, sorting results in 2-21
averages, moving 3-9
AVG function 2-25

B
BETWEEN predicate 2-15
BREAK BY subclause of ORDER BY clause 2-23
outer join with 5-15

calculations
aggregate functions 2-25
BREAK BY subtotals 2-23
datetime functions 3-25
RISQL display functions 3-1
with subqueries 4-9
CASE expressions
in comparison queries 4-5
in macros 6-3
with NTILE function 3-19
cases, tracked by technical support xvi
change in share, macros that calculate 6-15
Index

characters
  literals 2-7
  patterns 2-17
  wildcard 2-17
child queries, See subqueries
Class table, Aroma database 1-4, A-4
classes, See SELECT statement
COALESCE function 5-13
columns
  aliases 2-21, 2-27
  in Aroma tables A-5 to A-19
  selecting 2-5
comparison macros 6-11
comparison operators, list of 2-13, 2-38
comparison predicates 2-15
  ALL 2-15, 4-21
  ANY 2-15, 4-25
  BETWEEN 2-15
  EXISTS 2-15, 4-23
  IN 2-15
  in subqueries 4-21
  IS NOT NULL 2-15
  IS NULL 2-15
  LIKE 2-15
  list of 2-38
  NOT EXISTS 4-23
  SOME 2-15, 4-25
comparison queries 4-1 to 4-26
comparisons, market share 6-13
compound conditions 2-9, 2-11
  compound 2-9, 2-11
  correlation 4-13
  connectives, logical 2-9
control breaks 3-7
conventions
  syntax diagrams xii
  syntax notation xi
correlated subqueries 4-13
correlation names
  derived tables 4-7
  tables 4-13
COUNT function 2-25
CREATE MACRO command 6-3
CREATE TEMPORARY TABLE command 6-18
CREATE VIEW command 6-17
cross-references, as correlation
  conditions 4-13
  CUME function 3-5
  cumulative totals
    calculating 3-5
    resetting 3-7
  Customer Support Center xv
D
data list A-14
databases
  Aroma 1-2 to 1-7, A-1 to A-19
    decision-support 1-3
  DATEADD function 3-25
  DATEDIFF function 3-27, 5-5
  DATENAME function 3-25
dates, incrementing or decrementing 3-25
datetime scalar functions 3-25
Deal table, Aroma database A-16
DEC function 2-37
decision-support data analysis
  introduction 1-1
  key concepts 1-10
decrementing dates 3-25
derived column names 4-7, 4-9
derived tables
  produced by joins 5-3
  produced by subqueries 4-7, 5-21, 5-23
DESC keyword 2-21
descending order, sorting results in 2-21
dimension tables, Aroma database
  1-4, A-4, A-15
display functions, RISQL 3-3
  CUME 3-5
  in views 6-17
  MOVINGAVG 3-9
  MOVINGSUM 3-11
  NTILE 3-17
  order-dependent 6-17
  RANK 3-13
  RATIOTOREPORT 3-23
  resetting results of 3-7
  TERTILE 3-21
DISTINCT keyword
  in select list 2-5
  in set functions 2-25
division operator 2-33
Index

documentation
  list of Red Brick Systems ix
  support xvii

E
  e-mail addresses, for Red Brick Systems xv
  embedded macros 6-5
  equijoins, inner 5-3
  EXCEPT operator
    example 5-23
    syntax 5-17
  EXISTS predicate 2-15, 4-23
  explicit tables 2-3
  expressions, as cross-references 4-15
  EXTRACT function 3-29, 4-23

F
  fact tables, Aroma database 1-6, A-12, A-19
  fact-to-fact joins 5-12
  foreign key references 1-7, A-13
  FROM clause
    basic syntax 2-3
    joins in 5-3
    subqueries in 4-7
  full outer joins 5-11, 5-15
  functions, See display functions,
    RISQL functions, scalar functions

G
  generalized macros 6-7
  GROUP BY clause 2-29, 2-31

H
  HAVING clause 2-35

I
  IN predicate 2-15
  incrementing dates 3-25
  indexes, STAR A-13
  inner equijoins 5-3
  inner queries 4-7
  INSERT INTO SELECT statement 6-19
  intermediate tables, See derived tables
  INTERSECT operator
    examples 5-19, 5-21
    syntax 5-17
  IS NULL, IS NOT NULL predicates 2-15

J
  joins
    dimensions and facts 2-19
    equijoins 5-3
    fact-to-fact 5-12
    FROM clause 5-5
    full outer 5-15
    inner 5-3
    introduction to 5-3
    left outer 5-13
    named-columns 5-5
    natural 2-19, 5-5
    non primary key–foreign key 3-27
    outer 5-11
    self 5-9
    WHERE clause 2-19

K
  keywords, SQL
    ALL 2-15, 4-21
    AND 2-9
    ANY 2-15, 4-25
    AS 2-27
    ASC 2-21
    AVG 2-25
    BETWEEN 2-15
    BREAK BY 2-23
    COUNT 2-25
    CUME 3-5
    DATEADD 3-25
    DATEDIFF 3-25, 3-27
    DATENAME 3-25
    DESC 2-21
    DISTINCT 2-5, 2-25
    EXCEPT 5-17
    EXISTS 2-15, 4-23
    EXTRACT 3-29
    FROM 2-3
    GROUP BY 2-29, 2-31
    HAVING 2-35
    IN 2-15
    in syntax diagrams xiv

SQL Self-Study Guide IX-3
Index

keywords, SQL, continued
  INSERT INTO  6-19
  INTERSECT  5-17
  IS NULL  2-15
  LIKE  2-15
  MAX  2-25
  MIN  2-25
  MOVINGAVG  3-9
  MOVINGSUM  3-11
  NOT  2-9
  NTILE  3-17
  OR  2-9
  ORDER BY  2-21
  RANK  3-13
  RATIOTOREPORT  3-23
  RESET BY  3-7
  SELECT  6-19
  SOME  2-15, 4-25
  SUBSTR  6-13
  SUM  2-25
  SUPPRESS BY  2-37
  TEMPORARY  6-18
  TERTILE  3-21
  UNION  5-17
  WHEN  3-15
  WHERE  2-7

left outer joins  5-11, 5-13
LIKE predicate  2-15, 2-17
Line_Items table, Aroma database  A-18
literals, character and numeric  2-7
logical conditions, compound  2-11
logical connectives
  AND  2-9
  list of  2-38
  NOT  2-9
  OR  2-9

macros, RISQL
  comparisons  6-11
  embedded  6-5
  generalized  6-7
  multi-parameter  6-8
  naming conventions  6-3
  parameterized  6-7
  share comparisons  6-13
market share
  calculating changes in  6-15
  macros that calculate  6-13
  subqueries that calculate  4-9
Market table, Aroma database  1-5, A-6
matching character patterns  2-17
MAX function  2-25
moving averages  3-9
moving sums  3-11
MOVINGAVG function  3-9
MOVINGSUM function  3-11
multi-part primary keys  1-7
multi-star schemas  A-14

named-columns joins  5-5
natural joins  2-19, 5-5
negative operator (-)  2-33
NOT connective  2-9
NOT EXISTS predicate  4-23
notation conventions  xi
NTILE function
  inside a CASE expression  3-19
  ranking values in groups  3-17
NULL comparison predicate  2-15
NULLs
  ignored by set functions  2-25
  removing with SUPPRESS BY clause  2-37
numeric literals  2-7
### Index

**O**
- operators
  - arithmetic 2-33
  - comparison 2-13
  - UNION, EXCEPT, and INTERSECT 5-17
- OR connective
  - in conditions 2-9
  - versus UNION operator 5-16
- ORDER BY clause
  - BREAK BY subclause 5-15
  - referencing column aliases 5-17
  - required for RISQL display
  - functions 3-3
  - RESET BY subclause 3-7
  - subclauses 3-7
  - syntax 2-21
- order of precedence
  - arithmetic operators 2-33
  - logical connectives 2-9
  - parentheses 2-11, 2-33
- order-dependent display functions 6-17
- Orders table, Aroma database A-18
- outer joins 5-11
- outer queries 4-7
- outer references, See cross-references

**P**
- parameterized macros 6-7
- parent queries 4-7
- parentheses, order of precedence 2-33
- percentages
  - calculating with
    - RATIOTOREPORT 3-23
    - change in share 6-15
    - monthly, yearly 4-17
- Period table, Aroma database 1-4, A-8
- positive operator (+) 2-33
- precedence, order of 2-9
- predicates
  - ALL 2-15
  - ANY 2-15
  - BETWEEN 2-15
  - comparison 2-13, 2-15
  - EXISTS 2-15
  - IN 2-15
  - IS NOT NULL 2-15
  - IS NULL 2-15
  - LIKE 2-15, 2-17
  - SOME 2-15
  - wildcard characters 2-17
- primary keys 1-7, A-13
- Product table, Aroma database 1-4, A-4
- programmed breaks 3-7
- Promotion table, Aroma database 1-5, A-10
- purchasing schema, Aroma database A-14

**Q**
- qualified column names 4-15
- query expressions
  - CREATE VIEW statements 6-17
  - flexibility of 4-7
  - FROM clause 4-7
  - SELECT statements 2-3
  - UNION, INTERSECT, and EXCEPT operators 5-17
- query processing, order of 2-29
- query rewrite system 2-29

**R**
- RANK function 3-13
- ranking
  - the top ten 3-15
  - values as high, middle, low 3-21
  - values into unequal groups 3-17
- ratios, calculating 3-23
- RATIOTOREPORT function 3-23
- Red Brick Vista option 2-29, 3-3
- RESET BY subclause 3-7
- resetting cumulative totals 3-7
- retail schema, Aroma database 1-2, A-2
- right outer joins 5-11
Index

RISQL functions
See also scalar functions and display functions
  AVG 2-25
  COALESCE 5-13
  COUNT 2-25
  CUME 3-5
  DATEADD 3-25
  DATEDIFF 3-27
  DATENAME 3-25
  DEC 2-37
  display functions 3-3 to 3-23
  DISTINCT 2-25
  EXTRACT 3-29
  MAX 2-25
  MIN 2-25
  MOVINGAVG 3-9
  MOVINGSUM 3-11
  NTILE 3-17
  RANK 3-13
  RATIOREPORT 3-23
  set functions 2-25
  STRING 2-33, 5-5
  SUBSTR 6-13
  SUM 2-25
  TERTILE 3-21
RISQL macros, See macros, RISQL rows, selecting 2-7
running totals, calculating 3-5

S
Sales table, Aroma database 1-7, A-12
sample database, Aroma 1-2, A-2
scalar functions
  COALESCE 5-13
  DATEADD 3-25
  DATEDIFF 5-5
  DATENAME 3-25
  DEC 2-37
  EXTRACT 3-29, 4-23
  STRING 2-33, 5-5
  SUBSTR 6-13
scalar subqueries 4-11
schemas
  Aroma database 1-3, A-3
  multi-star A-14
segments, for Aroma Sales table 1-7, A-13

select expressions, See query expressions
SELECT statement
  BREAK BY subclause 2-23, 5-15
  FROM clause 2-3
  joins in 5-5
    subqueries in 4-7
  GROUP BY clause 2-29, 2-31
  HAVING clause 2-35
  ORDER BY clause 2-21, 5-15
  processing order 2-29
  RESET BY subclause 3-7
  summary of syntax 2-38
  SUPPRESS BY clause 2-37
  WHEN clause 3-15, 3-19
  WHERE clause
description 2-7
  join specification in 2-19
  subqueries in 4-19
  select-list subqueries 4-11
  self-joins 5-9
  set functions 2-25
  share calculations
    macros for 6-13
    subqueries for 4-9, 4-17
  shared dimensions, in outer join
    queries 5-14
  SOME predicate 2-15, 4-25
  sorting result sets 2-21
  spaces, removing with SUPPRESS BY clause 2-37
  STAR indexes A-13
Store table, Aroma database 1-5, A-6
STRING function, for truncating numeric values 2-33, 5-5
subqueries
  comparison predicates in 4-21
  correlated 4-13
  FROM clause 4-7
  select-list 4-11
  spreadsheet-style comparisons 4-5, 4-11
  WHERE clause 4-19
SUBSTR function 6-13
subtotals, calculating 2-23
subtraction operator 2-23
SUM function 2-25
sums
  calculating moving 3-11
  set function 2-25
Supplier table, Aroma database A-16
support
documentation xvii
technical xv
SUPPRESS BY clause 2-37
syntax diagrams
conventions for xii
keywords in xiv
metavariables in xiv
syntax notation xi
system tables, joining 5-7

T
table expressions, See query expressions tables
See also derived tables
aggregate 1-3
Aroma database 1-2 to 1-7, A-2 to A-19
joining 2-19, 5-3
system 5-7
temporary 6-19
technical support xv
temporary tables, creating 6-19
TERTILE function 3-21
top-ten rankings 3-15
troubleshooting, general problems xvii

U
unary operators 2-33
UNION, INTERSECT, and EXCEPT operators
examples 5-16 to 5-23
syntax 5-17
UNION versus OR 5-16

V.views, creating and selecting from 6-17

W
WHEN clause
example with NTILE 3-19
element with RANK 3-15
WHERE clause 2-19
subqueries 4-19
syntax 2-7
versus HAVING clause 2-35
wildcard characters 2-17
World Wide Web address, for Red Brick Systems xv

Z
zeroes, removing with SUPPRESS BY clause 2-37
Index
RED BRICK®

USA SALES OFFICES
1040 Crowne Point Parkway, Suite 250, Atlanta, GA 30338 +1 770 804 2440
2215 York Road, Suite 409, Oak Brook, IL 60521 +1 630 472 8500
1120 Avenue of the Americas, 4th Floor, New York, NY 10036 +1 212 626 6815
5314 Arapaho Road, Dallas, TX 75248 +1 972 702 1750
2010 Corporate Ridge, 7th Floor, McLean, VA 22102 +1 703 883 9310

UK SALES OFFICE
Red Brick Systems UK Ltd., 45 Berkeley Square, Mayfair, London W1A 1EB
United Kingdom +44 171 290 8373

AUSTRALASIA HEADQUARTERS
Red Brick Systems Australasia Pty. Ltd., Level 20, 99 Walker Street,
North Sydney, NSW 2060 Australia +61 02 9911 7744

JAPAN HEADQUARTERS
Red Brick Japan Co. Ltd., Level 16 Shiroyama Hills, 4-3-1 Toranomon,
Minato-ku, Tokyo 105 Japan +81 3 5403 4638